CHARACTERIZING SYSTEM FAILURES IN COMMERCIAL AND OPEN SOURCE SYSTEMS

BY

ZUONING YIN

DISSERTATION

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2012

Urbana, Illinois

Doctoral Committee:

Professor Yuanyuan Zhou, Chair & Director of Research
Professor Matthew Caesar
Professor Geoffrey Voelker, University of California, San Diego
Professor Chengxiang Zhai

Abstract

With the advance of technology, current systems are becoming much more powerful in computation, much faster in data transfer and much more abundant in data storage. However, what have been left behind are the system reliability and manageability. Current systems still fail quite often in the field. Understanding the characteristics of system failures is a prerequisite to come up with effective solutions to address these system problems. This thesis focuses on the failures introduced by incorrect bug fixes (a.k.a, buggy patches) and configuration errors.

Bug fixing is done by humans, therefore it can also introduce mistakes, which are incorrect fixes. These incorrect fixes not only further aggravate the damage to end users, but also poison software vendors' reputation. Therefore, we did one of the most comprehensive characteristic studies on incorrect bug-fixes from four large operating system code bases, including a commercial OS project. We studied the ratio and impact of incorrect fixes, and found incorrect fix is a significant problem that requires special attention. We also studied the common patterns of mistakes made during bug fixing that can be used to alert the programmers as well as to design detection tools to catch these incorrect fixes. We finally studied the code knowledge of developers and found inadequate code knowledge may increase the chance of incorrect fixes.

Configuration error (i.e., misconfigurations) is another dominant cause of system failures. Unfortunately, the characteristics of misconfigurations have been rarely studied in the past. Therefore, we took the initiative to conduct a real-world misconfiguration characteristic study. We studied a total of 546 misconfiguration cases, including 309 cases from a commercial storage system deployed at thousands of customers and 237 cases from four widely used open source systems (CentOS, MySQL, Apache HTTP Server, and OpenLDAP). Our study covers several dimensions of misconfigurations, including types, causes, impact, and system reactions. Some of our major findings include: 1) a majority of misconfigurations are due to mistakes in setting configuration parameters; however, non-parameter mistakes are still sizable. 2) 38.1%$\sim$53.7% of parameter mistakes are caused by illegal parameters that clearly violate some format or rules, motivating the use of an automatic checker to detect them. 3) a significant percentage (12.2%$\sim$29.7%) of parameter-based mistakes are due to inconsistencies between different parameter values.

To Father and Mother.

Thank you for giving birth to me.

You raise me up so that I can stand on mountains.

You raise me up so that I can walk on stormy seas.

You teach me to be a good person,

to be a man of integrity,

to be a man of compassion,

to be a man who can appreciate the beauty of the world and understand the essence of life.

When I fall down, you lift me up.

When I get lost, you show me the way.

When I am afraid, you give me encouragement.

When I feel happy, you feel much happier than me.

Without you, there is no me, nor this thesis.

Loving you, Father and Mother.

I will always pursue my dream and keep my faith,

to be a man that you can be proud of.

Acknowledgments

Special thanks to my advisor Professor Yuanyuan Zhou. Thank you for your support and instructions. You teach me many invaluable things. There are two things that I will never forget. The first is that one should be visionary, should think deep and long. The second is that the life is full of rejections, just take them and keep fighting.

Thanks to my beloved, Qiuqing, for always being at my side, for loving me, caring for me, encouraging me when the primroses bloom, when the grasshoppers sing, when the leaves turn yellow, when the snow flakes fly. You paint my life with color and fill my heart with joy.

Thanks to all my dear friends, Yan, Weiwei, Ding, Yuntao, Chao, Soyeon, Chengdu... Sorry that there are too many so that I cannot name them all. You guys make my life different.

Also thanks to all the other professors in my committee, Professor Matthew Caesar, Professor Geoffrey Voelker and Professor Chengxiang Zhai. Thanks for the kind help on my thesis.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

With the advance of technology, current systems are becoming much more powerful in computation, much faster in moving bits and much more abundant in data storage. However, what have been left behind are the system reliability and system manageability. Current systems (especially enterprise systems) are not only non-trivial to manage, but still fail quite often in the field. The stories [11, 37, 47, 89, 21, 63, 18, 7, 31, 14, 67] that relate to crashes, hangs, data corruptions or large-scale system outage can be easily found in the headline of some technical news.

As one recent example [11] , on February 16th 2010, a small ISP network (called Supronet) performed a configuration change to shift traffic from one of its links to another ISP. The modification resulted in routing updates that have a very long AS Path. Though these updates could be correctly handled by the MikroTik routers deployed in Supronet. Unfortunately, Cisco routers contained a bug that would cause them to reboot when receiving a long AS Path. Worse still, after rebooting, Cisco routers would try to receive those "unexpected" updates again and reboot again, triggering continuous oscillations. While Supronet, which is in the Czech Republic, performed this configuration change around midnight, it was mid-day/early evening in east Asia and U.S. when the fault occurred. The result was a hundred-fold increase in instability, traffic loss, and outages affecting nearly every country in the world.

Therefore, it is critical to find solutions to deal with system failures, considering that they not only bring huge financial costs to both system vendors and users [63, 7], but sometimes lead to irreversible damages such as human causality [1]. With the trend of cloud computing, the responsibility of maintaining desirable system reliability and manageability will be largely shifting to the cloud service providers and cloud infrastructure providers. On one hand, the clients are becoming thinner and thinner and they will have much less to worry about. But, on the other hand, the systems inside cloud will become even more complex. Then it will be more difficult to manage these cloud systems and to guarantee their reliability. This shift will definitely create new challenges for the whole industry to fight against system failures.

To prevent systems from failing, it is important to first have a comprehensive understanding on the characteristics of the failures. Generally, system could fail due to three major causes: hardware faults,

software bugs and operator errors.

Hardware faults are still a major issue that causes system to fail especially when we are pushing the limits of circuit for better performance. However, with the advance of technology, the current hardware is much more reliable than it was several tens of years ago and there have already been many systematic and mature solutions (e.g., hardware verification [53], redundant design [88], error correcting code, fault injection [46], etc.) to handle these faults. Therefore, we will not discuss the system failures due to hardware faults in this thesis.

With the dip in hardware faults, the reliability of a system is more and more affected by the other causes. One of them is software bug. Software failures greatly reduce system dependability. Since software programming involves great human efforts, as software becomes more and more complex, it is inevitable that human will make mistakes and introduce bugs into the system. Understanding the characteristics of bugs can help us to explore more effective software testing, debugging and diagnosis techniques and come up with better software engineering methods to minimize the number of bugs that escape into production runs. A substantial amount of research efforts have been spent with the outcome of a series of insightful bug characteristic studies [32, 94, 95, 57, 60, 104]. In this thesis, we will not try to conduct yet another general bug characteristic study. Instead, we will dive in a sub-domain of software bugs that are both important and not well studied by looking from another perspective. These are the bugs that happen during bug fixing (i.e. incorrect fixes). These incorrect fixes are even worse than the original bugs, since they further aggravate the damage to end users and poison software vendors' reputation.

Unlike software bugs that are mostly introduced during developing (or fixing) the system, the other major cause to system failures is introduced while using the system. They are called operator errors. For example, a user can accidentally turn off his computer by mistake or misconfiguring the system to let it malfunction. Operator errors are very challenging to handle since users can use the system in very diversified ways. It usually takes a lot of efforts to diagnose an operator error in the field. Therefore, Understanding the characteristics of operator errors is rather crucial for us to design better tools to detect and diagnose them, or even give us insights on how to improve system design to avoid operator errors fundamentally. Though operator errors are of various types, a major source of operator errors come from misconfigurations. Misconfigurations refers to that a user configures, arranges, or organizes the system (or different system components) in a wrong way which leads to the malfunction or failures of the system. Though people had studied techniques that can handle certain types of misconfigurations [103, 25, 101, 99, 24, 110, 92, 54, 93, 50] in the literature, there has not been a comprehensive characteristic study on misconfigurations yet. In this thesis, we will try to take the initiative to conduct such a study.

1.1 The Characteristic Study on Incorrect Fixes

As a man-made artifact, software suffers from various errors, referred to as software bugs, which cause crashes, hangs or incorrect results and significantly threaten not only the reliability but also the security of computer systems. Once a bug is discovered, developers usually need to fix it. In particular, for bugs that have direct, severe impact on customers, vendors usually have to release timely patches as soon as possible in order to minimize the amount of system unavailable time.

Unfortunately, since bug fixing is also done by human, human mistakes are inevitable. Some fixes either do not fix the problem completely or can even introduce new problems. For example, in April 2010, McAfee released a patch which incorrectly identified a critical Windows system file as a virus [31]. As a result, after applying this patch, thousands of systems refused to boot properly, had lost their network connections, or both. In order to compensate the customers who had suffered from this buggy patch, besides making a public apology on its website, McAfee agreed to reimburse the victims for the costs to repair their computers. In 2005, Trend Micro also released a buggy patch which introduced severe performance degradation [63]. The company received over 370,000 calls from customers about this issue and eventually spent more than $8 million to compensate customers. The above two incidents are not the only cases in recent history. As a matter of fact, there were many other similar events [18, 47, 67, 21] in the past which put the names of big companies such as Microsoft, Apple and Intel under spotlight.

We had also studied the security patches released by Microsoft in its security bulletin [8] since January 2000 to April 2010. Surprisingly, out of the total 720 released security patches, 72 of them were buggy when they were first released. They had either introduced new problems or had not fixed the old problem completely, resulting in updated patches. Considering the fact that these patches were expected to "heal" some severe problems and were typically applied automatically to millions of users immediately once released, such incorrect fixes, especially those introducing new problems, just like broken promise, would have enormous impacts and damages to end users as well as software vendors' reputation.

Few recent studies had been conducted on certain aspects of incorrect fixes [27, 90, 82, 44]. For example, Śliwerski *et al.* [90] studied the incorrect fix ratios in Eclipse and Mozilla. They found that large fixes are more error-prone and developers are easier to make mistakes during bug fixing on Friday. Purushothaman *et al.* [82] studied the incorrect fix ratio in a switching system from Lucent, but their focus was on the impact of one-line changes. Gu *et al.* [44] studied the incorrect fix ratio in three Apache projects, but they focused on providing a new tool to validate the patch.

While these studies have revealed some interesting findings, most of them focused more on incorrect fix ratios and studied only open source code bases. This thesis goes much beyond prior work, studying

both commercial and open source, large operating system projects, and investigating not only incorrect fix percentages, but also other characteristics such as mistake patterns during bug fixing, types of bugs that are difficult to fix correctly, as well as the potential reasons in the development process for introducing incorrect bug fixes.

To the best of our knowledge, this thesis presents one of the most comprehensive characteristic studies on incorrect fixes from large OSes including a mature *commercial* OS developed and evolved over the last 12 years and three open-source OSes (FreeBSD, OpenSolaris and Linux), The details of study, including the methodology, the major findings and the threats to validity are presented in Chapter 2.

1.2 The Characteristic Study on Misconfigurations

There are uncountable numbers of systems in the world that need to be configured in order to work normally and deliver the right functionality. However, most of these systems can not be configured automatically so that substantial human efforts have to be invested to do the configuration. As discussed previously, human errors will inevitably arouse during this process, which leads to misconfigurations.

Misconfigurations (i.e., configuration errors) have a great impact on system availability. For example, a recent misconfiguration at Facebook prevented its 500 million users from accessing the website for several hours [37]. Last year, a misconfiguration brought down the entire ".se" domain for more than an hour [89], affecting almost 1 million hosts.

Not only do misconfigurations have high impact, they are also prevalent. Gray's pioneering paper on system faults [43] stated that administrator errors were responsible for 42% of system failures in high-end mainframes. Similarly, Patterson et al. [79] also observed that more than 50% of failures were due to operator errors in telephone networks and Internet systems. Usually a majority of operator errors (or administrator errors) are misconfigurations [75, 71]. Besides being prevalent, configuration errors are also expensive to troubleshoot. Kappor [48] found that 17% of the total cost of ownership of today's desktop computers goes towards technical support and a large fraction of that is troubleshooting misconfigurations.

To ultimately solve the misconfiguration problem, substantial efforts and breakthrough in multiple related directions, including misconfiguration detection, misconfiguration diagnosis, misconfiguration tolerance and misconfiguration testing, are needed. Recently, the study on misconfigurations has been making applaudable progress with a series of excellent research work such as [99, 103, 25, 92, 54, 50]. To just name a few, Peerpressure [99] uses statistics methods on a large set of configurations to identify single configuration parameter errors. Chronus [103] periodically checkpoints disk state and automatically searches for

configuration changes that may have caused the misconfiguration being troubleshooted. Confaid [25] uses data flow analysis to trace the configuration error back to a particular configuration entry. AutoBash [92] leverages a speculative OS kernel to automatically try out fixes from a solution database in order to find a proper solution for a configuration problem. Kardo [54] adopts machine learning techniques to extract misconfiguration solutions out of users' UI sequences. Conferr [50] provides a very useful framework with which users can inject configuration errors of three types: typos, structural mistakes and semantic mistakes. This framework has already been used in Confaid [25].

Many of these work rely on some patterns or assumptions on misconfiguration errors which are not validated with a large amount of real world data. Furthermore, still some of fundamental questions about misconfigurations remain unanswered. These situations make the research on misconfigurtions like "sailing in the mist". Therefore, a comprehensive real world misconfiguration characteristic study could greatly benefit all the above research directions and tools. Moreover, understanding the major types and root causes of misconfigurations may help guide developers to better design configuration logic and requirements, and testers to better verify user interfaces, thereby reducing the likelihood of configuration mistakes by users.

Unfortunately, in comparison to software bugs that have well-maintained bug databases and have benefited from many software bug characteristic studies [33, 60, 94, 95, 57, 104], a misconfiguration characteristic study is much harder, mainly because historical misconfigurations usually have not been recorded rigorously in databases. For example, developers record information about the context in the code for bugs, the causes of bugs, and how it was fixed; they also focus on eliminating or coalescing duplicate bug reports. On the other hand, the description of misconfigurations is user-driven, the fixes may be recorded simply as pointers to manuals and best-practice documents, and there is no duplicate elimination. As a result, analyzing and understanding misconfigurations is a much harder, and more importantly, manual task.

In this thesis, we take one of the first attempts to study real world misconfigurations in both commercial and open source systems, using a total of 546 misconfiguration cases. The commercial system that we chose is the COMP-A[1] storage system deployed at thousands of customers. It has a well-maintained customer issue database. The open source systems that we chose include the widely used system software CentOS, MySQL, Apache and OpenLDAP. We carefully examined the 546 misconfigurations, and discussed our results with developers, support engineers and system architects of these systems to ensure that we correctly understood these cases. Our study was approximately 21 person-months of effort, excluding the help from several COMP-A engineers and open-source developers. The study has revealed some interesting findings which we will discuss in Chapter 3 in detail.

[1] We are required to keep the company's identity confidential.

Chapter 2

The Characteristics on Incorrect Fixes

2.1 Overview

The existence of software bugs creates a huge threat to the reliability of all kinds of systems. Many bug detection tools [2, 56, 59] had been developed in recent years to help reducing the number of bugs. Unfortunately, there are still an unlimited supply of bugs in software systems. To cure the buggy systems, bugs have to be fixed. Therefore, software developers usually allocate a significant amount of time in bug fixing and sometimes their work performance would even be evaluated by how many bugs they have fixed. Given the objectives and prevalence of bug fixing in software industry, it is a sad story that bug fixing could be incorrect. It can either introduce new problem or fail to fix the original problem completely. For customers, incorrect fixes are even harmful than the original errors. A single incorrect fix can be enough to cause severe damage to both customers and software vendors. Customers will be upset if the fix can't fix the problem, and they will be outrageous if the fix even breaks some functionality which was working previously. For software vendors, incorrect fixes not only let them suffer big financial loss, but also tarnish their image dramatically.

In Chapter 1, we had already discussed the importance of studying the characteristics of incorrect fixes. While there is one question that needs to be discussed first: what are the reasons for incorrect fixes?

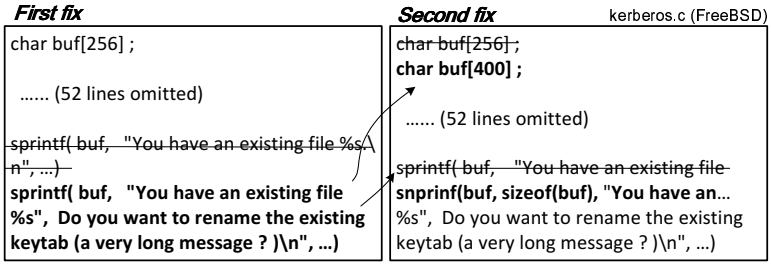

Figure 2.1: **An incorrect fix example from FreeBSD.** A part of the first fix appended a console message with some additional information, unfortunately introducing a buffer overflow (The added lines are in bold while the deleted lines are crossed out).

Mistakes in bug fixes may be caused by many possible reasons. First, bug fixing is usually under very

tight time schedule, typically with deadlines in days or even hours, definitely not weeks. Such time pressure can cause *fixers* [1] to have much less time to think cautiously, especially about the potential side-effects and the interaction with other parts of the system. Similarly, such time pressure does not leave enough time for testers to conduct thorough regression tests before releasing the fix. Figure 2.1 shows a real world example from FreeBSD, the original bug fix appended a log message with additional information. Unfortunately, the fixer did not pay attention to the buffer length that was defined 52 lines upwards in the same file and introduced a buffer overflow mistake.

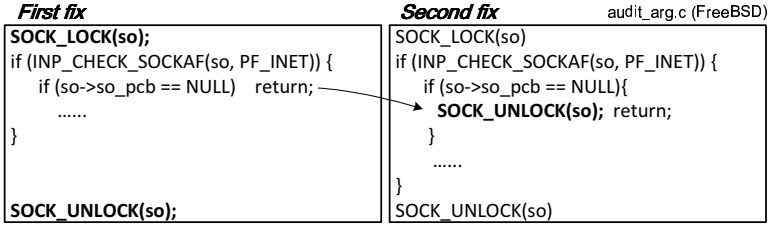

Figure 2.2: **An incorrect fix example from FreeBSD.** The first fix tried to fix a data race bug by adding locks, which then introduced a deadlock as it forgot to release the lock by calling *SOCK_UNLOCK* before *return*.

Second, bug fixing usually has a narrow focus (e.g., just removing the bug) comparing to general development. As such, the fixer regards fixing the target bug as the sole objective and the major accomplishment to be evaluated by his/her manager. Therefore, he/she would pay much more attention to the bug itself than the correctness of the rest parts of the system. Similarly, such narrowly focused mindset may also be true for the testers: tester may just focus on whether the bug symptom observed previously is gone, but forget to test some other perspectives, in particular how the fix interacts with other parts and whether it introduces new problems. As shown in Figure 2.2, the fixer just focused on removing the data race bug by adding locks. Though the data race bug was indeed removed, the fix unfortunately introduced a new bug: a deadlock. This deadlock was obviously not caught by the reviewers and not discovered during regression testing.

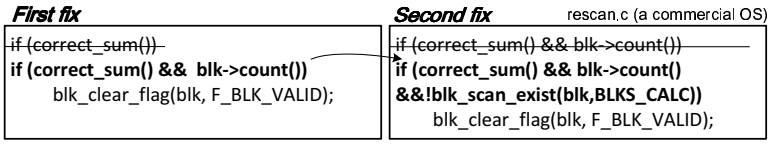

Figure 2.3: **An incorrect fix that hadn't fixed the problem completely.** *This example is from the large commercial OS we evaluated.* The first fix tried to address a semantic bug by modifying the *if* condition. Unfortunately, the revised condition was still not restrictive enough.

[1] we will refer the developer who fixes the bug as the "fixer" in the rest of the thesis.

Importance of Incorrect Fixes (Chapter 2.4, 2.5)	Implications
At least 14.8%∼24.4% of examined fixes for post-release bugs are incorrect. 43% of the examined incorrect fixes can cause crashes, hangs, data corruptions or security problems.	Although the ratio of incorrect fixes is not very high, the impact of the incorrect fixes indicate that the problem of incorrect fixes is significant and worth special attention.
Among common types of bugs, fixes on concurrency bugs (39% of them) are most error-prone, followed by semantic bugs (17%) and then memory bugs (14%).	Developers and testers should be more cautious when fixing concurrency bugs.
Incorrect fixes to Concurrency bugs (Chapter 2.6)	**Implications**
Fixes on data race bugs can easily introduce new deadlock bugs or do not completely fix the problem.	The synchronization code added for fixing data races need to be examined in more detail to avoid new deadlock. Knowing all the access locations to the shared objects is the key to fix data race completely.
Fixes to deadlock bugs might reveal bugs which were hidden by the previous deadlock.	Fixers need to further examine the path after deadlock in case there are some bugs hidden due to the existence of the deadlock.
Incorrect fixes to Memory bugs (Chapter 2.6)	**Implications**
Fixing buffer overflows by statically increasing the buffer size is still vulnerable to future overflows.	It is better to use safe string functions (e.g., *snprintf*) or bound checking to fix buffer overflow.
Fixing memory leaks can introduce dangling pointer bugs when freeing the memory without nullifying the pointer, and memory corruption when freeing something that should not be freed, or do not solve the problem completely when forgetting to free the members of a structure.	It is good to nullify the pointer after freeing the memory. It is also important to clearly understand what and when should be freed to avoid overreaction. Fixers should remember to free the structure members when freeing a complex structure to avoid an incomplete fix.
Human reasons to incorrect fixes (Chapter 2.7)	**Implications**
Comparing to correct fixes, the developers who introduced incorrect fixes have less knowledge (or familiarity) with the relevant code. *27%* of the incorrect fixes are even made by fixers who previously had never touched the files involved in the fix.	Code knowledge has influence on the correctness of bug fixes. It is dangerous to let developers who are not familiar with the relevant code to make the fix.
Interestingly, in most of the cases, the developers who are most familiar (5∼6 times of the actual fixers) with the relevant code of these incorrect fixes are still working on the project, but unfortunately were not selected to do the fixes.	Having a right software maintenance process and selecting the right person to fix a bug is important.
The code reviewers for incorrect fixes also have very poor relevant knowledge.	It is also important to select a developer who is familiar with the relevant code as the code-reviewer.

Table 2.1: **Our major findings of real world incorrect bug fix characteristics and their implications.** Please take our methodology and potential threats to validity into consideration when you interpret and draw any conclusions.

Third, the two factors above can be further exacerbated if fixers or reviewers are not familiar with the related code. While an ideal fixer could be someone with the most knowledge about the related code, in reality it may not always be the case to assign such person to be the fixer. Sometimes, it may be difficult to know who is the right person to do the fix. Even if such person is known, he/she may be busy with other tasks or has moved to other projects, and is therefore unavailable to perform the fix. Sometimes, it is due to the development and maintenance process. Some software projects have separate teams for developing and maintaining software (for example, the latter is usually done by a sustaining engineer team). All these

real world situations can lead to the case that the fixer does not have enough knowledge about the code he/she is fixing, and consequently increases the chance of an incorrect fix. This might help explaining the incorrect fix shown in Figure 2.3 from the commercial OS that we evaluated. When we measure the fixer's knowledge based on how many lines he had contributed to the file involved in the patch, we found that he had never touched this file in the past, indicating that he may not have enough code knowledge to fix the bug correctly.

Regardless what is the reason for introducing these errors during bug fixing and why they were not caught before release, their common existences and severe impacts on users and vendors have raised some serious concerns about the bug fixing process. In order to come up with better process and more effective tools to address this problem, we need to first thoroughly understand the characteristics of incorrect fixes, including:

- *what are the reasons for errors during bug fixing?* Are there any unique characteristics for bug fixing comparing to writing general code?
- *How significant is the problem of incorrect fixes?* More specifically, what percentages of bug fixes are incorrect? Comparing to code added for features, is code for fixing bugs more likely to introduce bugs? How severe are the problems caused by incorrect fixes? Are the answers to the above questions the same for both commercial and open source software projects?
- *What types of bugs are difficult to fix correctly?* Are some types of bugs just more difficult to fix correctly so that fixers, testers and code reviewers for these types of bug fixes should pay more attention and effort to avoid mistakes?
- *What are the common mistakes made in bug fixes?* Are there any patterns among incorrect bug fixes? If there are some common patterns, such knowledge would help alerting developers to pay special attention to certain aspects during bug fixing. Additionally, it may also inspire new tools to catch certain incorrect fixes automatically.
- *What aspects in the development process are correlated to the correctness of bug fixing?* For example, is fixers and reviewers' relevant knowledge related to incorrect fixes?

In this thesis, we did a comprehensive characteristic study on incorrect bug-fixes from large operating system code bases including Linux, OpenSolaris, FreeBSD and also a mature commercial OS (NetApp's OS for its storage systems) developed and evolved over the last 12 years. More specifically, from these four OS code bases, we carefully examined each of the 970 randomly selected fixes for post-release bugs and identified the incorrect fixes. To gain a deeper understanding of what types of bugs are more difficult to fix correctly

as well as the common mistakes made during fixing those bugs, we further sampled another set of 320 fixes on certain important types of bugs. We further studied the possible *human reasons* in the development and bug fixing process. Our major findings are summarized in Table 2.1. These findings provide useful guidelines for patch testing and validations as well as bug triage process. For example, *inspired from our findings, the large software vendor whose OS code was evaluated in our study is building a tool to improve its bug fixing and code review process.*

While we believe that the systems and fixes we examined well represent the characteristics in large operating systems, we do not intend to draw any general conclusions about all the applications. In particular, we should note that all of the characteristics and findings in this study are associated with the types of the systems and the programming languages they use. Therefore, our results should be taken with the specific system types and our methodology in mind.

In Chapter 2.2, we present the background of our work and the related work. We then discuss the methodology used in our study and threats to validity in Chapter 2.3. After that we present our detailed results on the incorrect fix ratio in Chapter 2.4. Then we further study which types of bugs are more difficult to fix in Chapter 2.5 and what common mistakes could be made in Chapter 2.6. After that we study the human factors which could lead to incorrect fixes in Chapter 2.7.

2.2 Background

Studying incorrect fixes As briefly discussed in Chapter 1.1, several previous studies [27, 90, 82, 44] had also studied incorrect fixes. Our work is complementary to these previous studies. First, we focus on large OS projects, while previous studies focused on certain types of applications. Second, we study both commercial and open source code bases, while previous work studied only either open source or commercial code bases. Third, previous studies more focused on measuring incorrect fix ratios, while we went much beyond and also studied what types of bug fixes are more error-prone, the common mistake patterns, as well as the possible human reason in the development process for introducing incorrect fixes.

Śliwerski *et al.* [90] proposed an effective way to automatically locate fix-inducing changes by linking a source code repository to a bug database. They studied the incorrect fix ratio in Eclipse and Mozilla and also found developers are easier to make incorrect changes on Friday. Purushothaman *et al.* [82] studied the incorrect fix ratio in a switching system from Lucent, but their focus was just the impact of one-line changes. Gu *et al.* [44] studied the incorrect fixes in three Java applications, while their focus is to present a tool which can be used to verify patches. Baker *et al.* defined incorrect fixes as *fix-on-fix* and visualized

the fix-on-fix rates of the different modules inside a switch system of AT&T in [27]. However, their work focused on visualization. Besides, Beattie *et al.* [28] pointed out security patches could contain bugs.

Human factors Human factors can play an important role in software quality. The influence of code knowledge on general code changes had been explored in [84, 68]. Mockus *et al.* [68] found that changes made by more experienced developers were less likely to induce failures. Rahman *et al.* [84] found file owner with higher knowledge is less associated with fix-inducing code. Our study focused on *bug fixes* and measured the knowledge of the fixers who made the incorrect fixes in commercial and widely used open source OSes. We also found knowledge has impact on the quality of fixes, which is complimentary to their results. We found 27% of the incorrect fixes are made by fixers with *zero* knowledge, suggesting there might be some flaws in the overall bug assignment process. Fritzy *et.al.* [41] studied whether a programmer's activity can be used to build a knowledge model about a code base. Some work [20, 65] also studied human factors for designing recommendation systems. Anvik *et al.* [20] suggested to assign fixer based on bug history. McDonald *et al.* [65] suggested to find the person who last modified the code. We proposed to assign fixer/reviewer based on code knowledge defined at line level. Besides, other aspects of human factors had also been studied. Meneely *et al.* [66] found that independent developer groups were more likely to introduce a vulnerability. Bird *et al.* [29] found that a binary might be more buggy if more developers are working on it. Nagappan *et al.* [70] studied the organizational structure and used it to build model to predict the failure proneness in Windows Vista. Latoza *et al.* [55] studied developers' typical activities and found that developers spend nearly half of their time fixing bugs. Aranda [23] studied which coordination patterns are essential to the solution of the bug.

Taming incorrect fixes There are different ways to solve the problem of incorrect fixes including predicting or isolating buggy changes [91, 52, 64, 109], patch validation [44, 96], automatic patching [80] and regression testing [87, 76]. Śliwerski *et al.* built a plug-in for Eclipse which shows the risk of changing a particular code location based on previous revision information. Kim *et al.* [52] also leveraged the historical source repository data to train models for predicting the correctness of a future change. McCamant *et al.* [64] compared operational abstractions generated from the old component and the new component to predict the safety of a component upgrade. Zeller *et al.* [109] proposed automated delta debugging to locate the bug introducing changes. ClearView [80] automatically generates patches without human intervention, which can reduce the chance of incorrect fixes. Tucek *et al.* [96] used the techniques of delta execution which can validate patches on-line efficiently. Besides, regression testing [87, 76] is also a common practice to ensure patches don't break the previously working functionalities. Our study discovered some incorrect fix patterns which are helpful for detecting/exposing/avoiding incorrect fixes. We studied what mistakes programmers

should be aware of during bug fixing, which are also useful to design new detection tools to detect errors in fixes. Besides, we also proposed a bug assignment process based on code knowledge, which is being implemented by a large software vendor.

2.3 Methodology

2.3.1 Software Projects under Study

System	LoC	SCM	Open Src?
The commercial OS	confidential	confidential	N
FreeBSD	9.97M	svn	Y
Linux	10.94M	git	Y
OpenSolaris	12.99M	hg	Y

Table 2.2: **The four OSes that our study uses.** SCM is the Software Configuration Management system each OS uses. We also list the SCM used by each code base.

Table 2.2 lists the four code bases we studied, including a commercial, closed-source OS from a large software vendor (NetApp) [2] and three open-source OSes (FreeBSD, Linux and OpenSolaris). We chose to study OS code because they are large, complex and their reliability is critically important. Additionally, as OS code is developed by many programmers, contains lots of components, uses a variety of data structures and algorithms, it could provide us a broad base to understand incorrect fix examples.

The four OSes have different architectures. The commercial OS is especially designed for high-reliability systems with many enterprise customers like big financial companies and government agencies. It has evolved for almost 12 years. Its bug tracking system keeps the links to corresponding changes and its source code repository also keeps the reversed links, which provides great convenience to our study. Additionally, its bug tracking system has a field to record the customer impact when a bug causes failure at customer side and is reported by customers. This field is not available in the bug tracking systems for open source projects and it is very useful since it reveals the bugs which have real impact in the field. The other three open-source OSes have different origins. FreeBSD originates from academia (Berkeley Unix). It is a large OS project with a long history. The project started in 1993 and currently is working on its 9.0 series. FreeBSD is known for embracing a different design from UNIX and also famous for its high reliability and security guarantee. OpenSolaris originates from a commercial OS. It is based on Solaris [13] which is a closed-source operating systems created by Sun Microsystems. Its origin is the UNIX System V Release 4 (SVR4) developed by Sun Microsystems and AT&T in the late 1980s. Linux completely originates from the open-source community.

[2] Due to confidentiality agreement, we can neither mention the commercial OS's name, SCM nor its LoC.

It is a leading server operating system, and runs the 10 fastest supercomputers in the world [6]. In addition, more than 90% of today's supercomputers run some variant of Linux [10]. We think the variety in data sources would help us find general software laws or interesting specificities.

These OSes usually have multiple branches (series) in their OS families. We focus on those branches which are both stable and widely deployed. For the commercial OS, we chose the branch which is most widely deployed. For FreeBSD, we chose FreeBSD 7 series. For Linux, we chose Linux 2.6 series. For Opensolaris, it has a different release model so we just studied the releases since its 2008.5 version.

In order to further preserve the privacy and reputation for the software vendor, we anonymized the results in Chapter 2.4, 2.5, 2.6 and 2.7. The four code bases will be just referred as A, B, C and D without the mapping information disclosed. We know that such anonymization may prevent us from making some interesting comparison between open source and commercial code bases, but fortunately we can still make many other findings like the ones summarized in Table 2.1.

2.3.2 Finding Incorrect Fixes

The definition of incorrect fix: A bug fix f_x is defined as an incorrect fix if there is another following bug fix f_y that fixes either a new problem introduced by f_x, or the original problem that was not completely fixed by f_x.

One thing that we want to emphasize is that we consider only fixes to bugs, not any general or non-essential changes [49] (e.g., feature addition or renaming). This is identified by checking whether a fix is associated with a bug report. This screening criteria is important to the fidelity of our study since bug reports often contain rich information which is important for us to understand the fix. It is hard to obtain a complete picture from the bug fix itself alone.

Unfortunately, the link between fixes and bug reports is not always systematically maintained [26]. For the commercial OS and OpenSolaris, the SCM (software configuration management) systems record the link between every bug report and every code change. However, for FreeBSD and Linux, such links are only documented voluntarily by developers in an unstructured way. To identify such links, we use a method similar to the methods used in [39, 90, 98]. The main idea is to leverage the verbal information in the bug reports or change logs to reconstruct the links. For example, developers may write "*the bug is fixed by change 0a134fad*" in a bug report. Then we can link the bug to the change 0a134fad.

After the above process, we will have a set of bug fixes linked with bug reports. We then randomly select a target number of bug fixes and then semi-automatically check whether each one is an incorrect fix or not. We call the process semi-automatic because we use a two-step process: first step automatically selects

potential incorrect fix candidates, while the second step manually verifies each candidate.

Two techniques are used in the first step to automatically identify potential incorrect fix candidates. First, we look at the *source code overlap* between changes. This technique is similar to the methods used in [90, 82]. If there is source code overlap between two changes, then the latter change may be made to correct the previous change. There are *explicit overlap* and *implicit overlap*. The examples in Figure 2.1 and Figure 2.3 demonstrate the cases for explicit overlap, where the *sprintf* in first fix was explicitly overwritten by the *snprintf* in the second fix. The example in Figure 2.2 illustrates the case for implicit overlap where the latter change only adds some code in the proximity of the previous change. In Figure 2.2, three new lines of code were added without deleting or overwriting the code from the first fix. Therefore, we check source code overlap as follow: if a latter change f_y overwrites or deletes the code written in the previous change f_x or f_y just adds code in the proximity (+/-25 lines) of f_x, we regard f_x as an incorrect fix candidate for manual examination. The second technique is to search for specific keywords in the bug report and change log of each fix that may suggest an incorrect fix. For example, if we find "*this patch fixed a regression introduced by the fix in Bug 12476*" in the bug report linked with f_y, we regard the fix in "Bug 12476" as an incorrect fix candidate. In general, we find the first technique to be more comprehensive.

Please note that the first step is only identifying candidates. We still need to manually examine each candidate, which is the unique challenge in our study. This step is the part in our study where we spent the most of the time. We examined all the relevant code related to each fix. In many cases, the relevant code has a much bigger code size than the fix alone, and can cover other files which are not included in the fix. We also examined the bug reports and change logs to get proof from developer's explanation. For some fixes, we even discussed with developers of these systems to ensure the correct understanding on them. Then based on all the evidences we got, we finally decided whether a fix is incorrect or not. Figure 2.4 showcases the whole process.

Also note that the first step may prune a few incorrect fixes out, especially for those incorrect fixes whose subsequent fixes are in a completely different location without any overlap at all (i.e. beyond the +/-25 lines proximity). But we expect such incorrect fixes are very rare as two subsequent fixes to the same problem usually has good location proximity in terms of code changes. And we did try to relax the proximity requirement to be "within" the same file but did not find more incorrect fixes.

2.3.3 The Target Bugs to Study

In this study we used two sets of bug fixes with different focuses.

Sample set 1: To get this sample set, we first randomly sampled a total of 2,000 bug fixes (500 from each

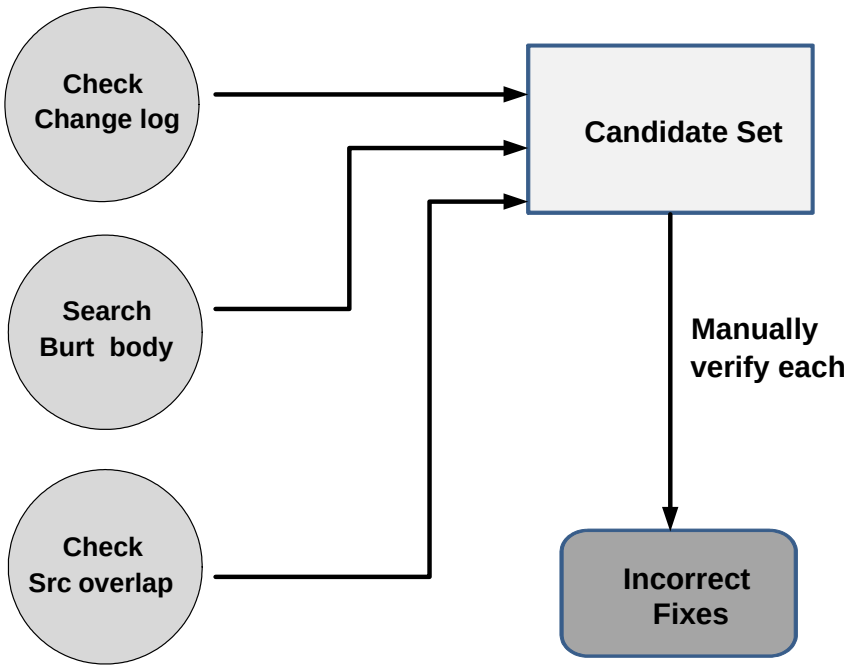

Figure 2.4: **The whole flow to get incorrect fix samples.**

OS) that are associated with bug reports. From these 2,000 bug fixes, we further selected ony those fixes to *post-release* bugs. Selecting such post-release bug fixes allows us to focus on fixes to bugs that made high impacts to both customers and vendors. Post-release bugs are selected after the random sampling instead of before the sampling because the manual effort in verifying all bug fixes would be too huge. We then used the process described previously to identify and study incorrect fixes.

Sample set 2: This sample set is used to further zoom in certain bug types observed in *sample set 1* whose fixes are most error-prone. Specifically, we chose to study the fixes to memory leak, buffer overflow, data race and deadlock bugs. However, it is difficult to reuse the bugs in *sample set 1*, since there are not statistically sufficient number of bug fixes for these types of bugs. Therefore, we deliberately sampled more bug fixes focusing on these four types. Specifically, we used all the related keywords to search for the bug fixes of a specific type. Keyword search is enough to get these bug fixes since there are only limited ways to name them. Then we randomly selected 20 from each type for each code base. In total, we sampled 320 fixes which provide us a richer base-set to study the incorrect fixed patterns. This set is only used in Chapter 2.5 and 2.6.

2.3.4 Measuring Code Knowledge

To understand why a programmer cannot fix a bug correctly, we also dive deeper into his/her knowledge about the relevant code. In this study, we measure code knowledge by checking the cumulative "authorship" of each line of code at a particular version, which can be systematically measured. From SCM, we obtain the authorship of each line for a file at a given version by using commands such as "svn annotate". Assume a developer d, a file F, a function f and a version v, we calculate code knowledge at two levels of granularity:

$$K\_File_{d,F,v} = \frac{\text{The LoC written by } d \text{ for } F \text{ at } v}{\text{The total LoC in } F \text{ at } v}$$

$$K\_Func_{d,f,F,v} = \frac{\text{The LoC written by } d \text{ for } f \text{ in } F \text{ at } v}{\text{The total LoC in the } f \text{ at } v}$$

Then we can calculate a developer's K_File and K_Func for each file and each function at a particular version. We use percentage as the unit of K_File and K_Func. For example, "$K\_File_{d,F,v}$=75%" means 75% of code lines in F at version v are written by d. d does not need to contribute all his code in one version and he may write these code lines in any version that is not later than v. Both fixers' and reviewers' knowledge are measured in our study in this way.

2.3.5 Threats to Validity

Real world empirical studies are all subject to validity problems, so is our study. Potential threats to the validity of this study are the representativeness of the selected software projects, the representativeness of the incorrect fix samples, our classification process and evaluation methodology.

Representativeness of software: Both commercial and open-source software are covered in this study. So we believe that we have a good coverage for at least OS code. We do not intend to draw any general conclusions about incorrect bug-fixes in all software, but some of the findings such as fixers and reviewers' knowledge would also apply to other applications.

Representativeness of bug fix samples: We studied only those bug fixes that could be linked to a bug report. The set of fixes F_n that cannot be linked were not covered, and some of our findings might not hold in F_n [26]. Fortunately, the results from the commercial OS and OpenSolaris are immune to this threat since every bug fix is linked to a bug report. Since the results from FreeBSD and Linux show a similar trend as the commercial OS and OpenSolaris, which may ease the concern on this threat for the other two OSes. As discussed earlier, there is also a potential problem in our automatic filtering process, i.e., we can potentially filter out an incorrect fix if its subsequent fix does not have any proximity in code location. Fortunately our exercises of relaxing the proximity constraint did not discover any more incorrect fixes, which indicates that

the amount of missed incorrect fixes should be very low.

Threats of manual classification: Our study involves manual classification on bug reports and fixes which can not be replaced by automatic techniques. Therefore, subjectivity is inevitable. However, we tried our best to minimize such subjectivity by using double verification. For every incorrect fix candidate, we examined all the information sources we could have, including source code, bug reports, change logs, etc. Besides, for some fixes we also discussed with developers of these systems to ensure the correct understanding on them. Since we manually examined each incorrect fix candidate and classify it as incorrect only if we have concrete evidence, we are confident that the number of false positives should be very low.

Limitation in measuring the knowledge: The way we measure code knowledge is relatively preliminary since we only want to check the fixers and reviewers' knowledge in a coarse-grain, qualitative way. A more sophisticated knowledge model might provide us more accurate results in Chapter 2.7, which remains as our future study.

2.4 Is Incorrect Fix Really a Significant Problem?

The ratio of incorrect fixes among all bug fixes and the impact of bugs introduced by incorrect fixes are important for us to accurately understand whether incorrect bug fix is a significant problem. As described in Chapter 2.3, we first randomly sampled 2,000 bug fixes from the four OSes (500 from each OS), among them 970 are fixes to post-release bugs. The reason we focus on fixes for post-release bugs is that those bugs hit the end users and generally have higher impact, hence software vendor and developers may spend more efforts in fixing them. On the other hand, the code base is usually not stable before release. Thus the ratio of incorrect fixes for before-release bugs may not represent the exact efforts that developers put into fixing bugs.

System	# of Post-release bug fixes	# of Incorrect fixes	Ratio
A	189	39	20.6%±3.0%
B	309	46	14.8%±2.9%
C	267	41	15.3%±2.6%
D	205	50	24.4%±3.7%

Table 2.3: **The ratio of incorrect fixes on post-release bugs in the four OSes.** A 95% confidence interval is used.

Table 2.3 shows the ratio of incorrect fixes is 14.8%∼24.4% among the four OSes. As discussed in Chapter 2.3.5, this is only a lower-bound estimation. Considering that the fixes on post-release bugs would be applied by a lot of customers and users, even the ratio can still have significant impact on them as well

as the software vendors. Since regression testing had already been applied before releasing the fixes, it also indicates general testing techniques may need to be tailored to be more effective in capturing the errors in patches.

System	d_{fix}	d_{whole}	d_{whole}/d_{fix}
A	171.4	6077.6	36x
B	840.6	4045.7	5x
C	561.2	6239.0	11x
D	712.7	3530.9	5x

Table 2.4: **The bug density measured in how many LoC contain a bug.** d_{fix} and d_{whole} represent the bug density in bug fixes and whole code base respectively.

We also compared the *bug density* in bug fixes with the whole code base. Table 2.4 shows this result. We found in all four OSes, the bug density in fixes is at least 5 times of the whole code base. In some OS project, the bug density in bug fixes can be *36* times of the whole code base. This indicates writing bug-fixing patches might be more error-prone. However, we used a simplified way to calculate the bug density. Therefore the results in Table 2.4 might not be accurate enough and should be taken with caution.

We further studied the impact of the bugs introduced by the examined incorrect fixes. We judge the impact based on the symptoms described in the bug reports. to see whether they could be "milder" or "more severe" than the original bugs. We found 14.0% of them introduced crash, 8.4% caused system to hang, 15.4% led to data corruption or data loss, 5.6% caused security problem, 7.0% degraded the performance, and 45.1% introduced incorrect functionality. Some bugs introduced are actually more severe than the original bugs. Moreover, for some bugs, they could even be incorrectly fixed for several times.

Finding: At least 14.8%∼24.4% of the bug fixes are incorrect. Moreover, 43% of the incorrect fixes resulted in severe bugs that caused crash, hang, data corruption or security problems.

Implication: Incorrect fix is indeed a significant problem that requires special attentions from software vendors.

2.5 Which Types of Bugs Are More Difficult to Fix Correctly?

In this section, we study which types of bugs are more likely to introduce incorrect fixes. We classified all the 970 sampled fixes into three categories based on the bugs they fix: memory bug, concurrency bug or semantic bug. Semantic bugs are those bugs that can be classified as neither memory nor concurrency bug and are usually application specific problems. This classification is adopted from previous literature [57, 61].

System	Concurrency	Memory	Semantic
A	4/13 (31%)	3/17 (18%)	32/159 (20%)
B	9/21 (43%)	5/44 (13%)	32/244 (13%)
C	7/19 (37%)	6/43 (14%)	28/205 (14%)
D	10/23 (44%)	5/30 (17%)	35/152 (23%)
Overall	30/76 (39%)	19/134 (14%)	127/760 (17%)

Table 2.5: **The number of incorrect fixes among all the fixes and the incorrect fix ratio for the three categories of bugs in the four OSes.**

Table 2.5 shows the ratio of incorrect fixes to each type of bug. In comparison, concurrency bugs have the largest incorrect fix ratio (39% overall), indicating concurrency bugs are the most difficult to fix. Semantic bugs and memory bugs have similar ratio, 17% and 14%, respectively.

We focus on studying concurrency bugs and memory bugs, while only providing some high level discussion on semantic bugs (Chapter 2.6.5). This is because semantic bugs are of very diverse root causes so that it is difficult to observe some general patterns from their fixes and the mistakes in the fixes. Also concurrency bugs and memory bugs usually have higher impact (e.g., causing crash or data corruption) than semantic bugs (e.g., causing functionality problem).

Bug Types and Their Percentages			
data race	33%	deadlock	29%
buffer overflow	8%	memory leak	6%
uninitialized read	4%	null pointer deref	4%

Table 2.6: **The most observed bug types among all the concurrency bugs and memory bugs being fixed incorrectly.** Only top six are shown.

To select the important types of bugs for a detailed study, we further zoomed into all the concurrency bugs and memory bugs that were fixed incorrectly to see which types are most observed. The result is shown in Table 2.6. Among all the bug types, data race (33%), deadlock (29%) are the top two types that have most incorrect fixes among concurrency bugs. Buffer overflow (8%) and memory leak (6%) are the top two types that have most incorrect fixes among memory bugs. Therefore, we just focused on the characteristics of bug fixes to these four types of bugs.

Table 2.7 further shows the ratio of incorrect fixes for these four types of bugs. The result is from 320 fixes only to these bug types (*sample set 2* mentioned in Chapter 2.3.3), where for each type we randomly sampled 20 fixes from each code base. We use this data set instead of the one used above because among the original 970 fixes (*sample set 1* mentioned in Chapter 2.3.3), there are not statistically sufficient number of bug fixes for these four types of bugs. In other words, in this study we deliberately sampled more bug fixes focusing on these four types. *sample set 2* provides us a richer base-set of incorrect fixes to conduct our case studies in Chapter 2.6. As indicated in Table 2.7, fixes to data race and deadlock are most error-prone, with

System	Bug types			
	race	deadlock	buf overflow	mem leak
A	9/20 (45%)	5/20 (25%)	2/20 (10%)	1/20 (5%)
B	11/20 (55%)	6/20 (30%)	1/20 (5%)	3/20(15%)
C	11/20 (55%)	8/20 (40%)	3/20 (15%)	0/20 (0%)
D	8/20 (40%)	9/20 (45%)	1/20 (5%)	4/20 (20%)
All	39/80 (49%)	28/80 (35%)	7/80 (9%)	8/80 (10%)

Table 2.7: **The number of incorrect fixes among the all the fixes and the incorrect fix ratio for the four important types of bugs from *sample set 2*.** The format is "incorrect/all sampled fixes (ratio)".

an incorrect fix ratio of 49% and 35% respectively, which is generally 4x∼6x of buffer overflow and memory leak.

Finding: Concurrency bugs are the most difficult (39%) to fix right. Among concurrency and memory bugs which were fixed incorrectly, the four most observed bug types are: data race, deadlock, buffer overflow and memory leak.

Implication: Developers and testers should be more cautious when fixing concurrency bugs. The allocation of fixing and testing resources could consider the types of bugs to be fixed.

2.6 Incorrect Fix Patterns

After understanding which types of bugs are more difficult to fix right, it would be interesting to study the common mistakes (patterns) when fixing a particular type of bugs and the consequence introduced by those incorrect fixes. Though bug fix patterns had already been studied in [51, 78], few had studied the patterns of incorrect fixes before. In this section, we use the incorrect fix examples got from *sample set 2*. Generally, we find there are two types of incorrect fixes: *incomplete fixes* and *introducing new problems*, while each type of bug also has its own incorrect fix patterns. We also discuss techniques to detect or reveal these mistakes by either extending current techniques or suggesting new approaches.

2.6.1 Fixing Data Races

The most common practice for fixing data race is to add synchronization primitives (e.g., locks) to guard shared resources with mutual exclusion. However, delivering a correct fix requires deep reasoning on all the side-effects of the newly added synchronization, which is often error-prone.

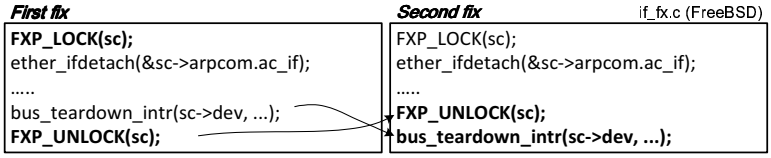

Figure 2.5: **Incorrect fix to a data race introduced a deadlock.** The function *bus_teardown_intr* cannot be called with lock held, otherwise deadlock will be introduced.

Specifically, adding locks might introduce deadlock. This incorrect fix pattern is observed in all the four code bases we evaluated and in 16.4% (6 out of 39) of the incorrect fixes to data race bugs. Figure 2.5 shows one of the examples. In the first fix, a lock *sc* was added to avoid a race. However, the function *bus_teardown_intr* is not supposed to be called inside the critical section, since it can lead to deadlock. Unfortunately, developers were not aware of this rule and made the incorrect fix. To fix this deadlock, *bus_teardown_intr* was moved out of the critical section.

Figure 2.2 (in Chapter 2.1) is another example of this pattern that fixing data race introduces deadlock. The fixer forgot to release the lock via *SOCK_UNLOCK* before a *return* statement therefore a deadlock happened.

Implications: When adding synchronization primitives, fixers need to make sure the newly added primitives (e.g., lock) will not introduce deadlocks with the existing synchronization code. This can be checked by extending deadlock detectors to only focus on the synchronization primitives newly added. Besides, lock and unlocks should be added in pairs along all the execution paths in the newly formed atomic region. This can be checked automatically by extending some existing path-sensitive bug detection tools such as RacerX [35] or PR-Miner [58] to only scan the code regions touched by the fix.

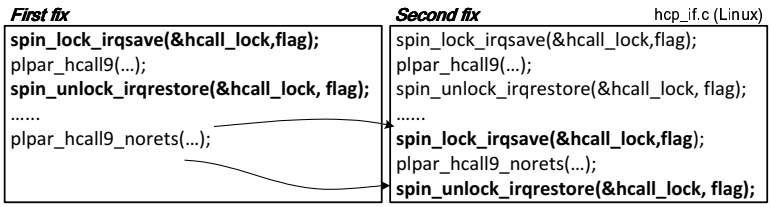

Figure 2.6: **Fix to a data race was not complete.** The first fix only added locks to protect function *plpar_hcall9*, while forgot to protect *plpar_hcall9_noret* (which contains the access to the same shared objects in *plpar_hcall9*).

Fixing data races could be incomplete such that not all the data races are fixed. This incorrect fix pattern is observed in three of the code bases we evaluated and in 10.2% (4 out of 39) of the incorrect fixes to data race bugs. For example, as shown in Figure 2.6, when adding locks, the developer forgets to lock all the

places she should lock.

Implications: For a complete fix to data race, it is important to know all the accesses to the shared objects which could race with each others. While this task might be daunting for manual efforts, incomplete fixes as shown in Figure 2.6 can be detected by extending techniques such as [36] or PR-Miner [58] with the focus on the newly added lock. We can design checkers to detect where the same shared objects are protected by lock in some paths while unprotected in some others [36], and make these checkers focus only on checking the patched code.

2.6.2 Fixing Deadlocks

To fix a deadlock, developers may either reverse the order of locks, or even drop some locks. However, these means need to be applied with caution.

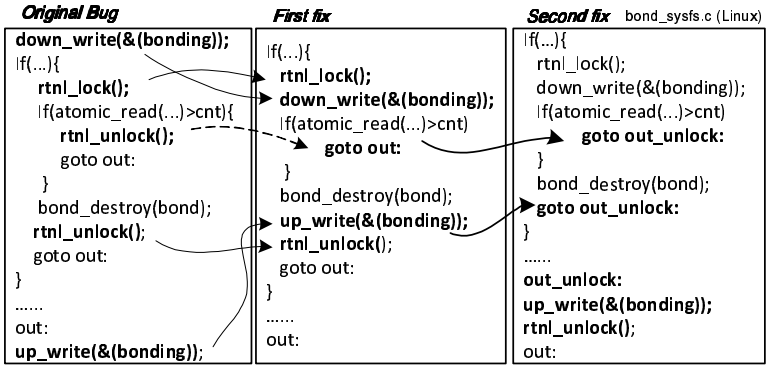

Figure 2.7: **Incorrect fix to a deadlock introduced a new deadlock.** The first fix reversed the order of locks to prevent deadlock, but forgot to release locks before taking a *goto* path.

Specifically, fixing deadlocks could still lead to deadlock bugs. This incorrect fix pattern is observed in three of the code bases we evaluated and in 14.3% (4 out of 28) of the incorrect fixes to deadlocks. Figure 2.7 shows such an example. The root cause of this incorrect fix is similar to the one in Figure 2.2. Therefore we can again extend some current path-sensitive bug detection tools to spot the deadlock.

Additionally, fixing deadlock may reveal some other bugs that were originally hidden by the deadlock, especially data race bugs. This incorrect fix pattern is observed in two of the code bases we evaluated and in 7.1% (2 out of 28) of the incorrect fixes to deadlocks. Though we only spotted 2 such cases, we think this is still an interesting pattern. Figure 2.8 shows one of the examples. There are two bugs in the original code: a deadlock caused by the wrong order of the two locks, and a data race caused by an unprotected shared variable in the second *for* loop. However, the data race is hidden by the existence of the deadlock since the execution would not even reach the second *for* loop due to the deadlock. The first fix exchanged the order

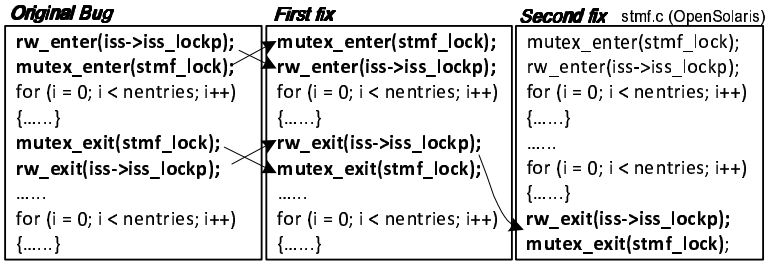

Figure 2.8: **A fix to a deadlock exposed a hidden data race bug.**

of two locks (*stmf_lock* and *iss_lockp*) and resolved the deadlock. However, it also enables the execution to proceed so that the data race is much easier to manifest.

Implications: The hang introduced by deadlock bugs might prevent some execution paths from being exercised thoroughly, which could make some bugs hidden in those paths difficult to manifest. After removing deadlock bugs, fixers should further test those execution paths.

2.6.3 Fixing Buffer Overflows

We also found some interesting incorrect fixes examples for memory bugs. However, since the total numbers of incorrect fixes to buffer overflows and memory leaks in *sample set 2* is not statistically high enough (7 and 8 respectively), we do not claim those examples are frequently observed incorrect fix patterns. However, we assume these examples could be common for the incorrect fixes to buffer overflows and memory leaks (shown in Section 2.6.4) if we can further enlarge our sample set.

Common techniques to fix buffer overflow include: a) restrict the length of the data which will be stored into buffer by using safe string functions (e.g., *snprintf*) or do bound checking; b) increase the buffer size statically from stack; c) allocate larger buffer dynamically from heap to replace a stack buffer.

Based on our observation, technique a) is usually safe and seldom introduces any further incorrect fixes since it eradicates the chance of a buffer overflow in the future. Precautions are needed for the use of bound checking. Bound checking is usually followed by *return* or *goto* statements for error handling. These statements may change the control flow of the original code and introduce some other mistakes.

Implications: The good practice to fix buffer overflow is to use safe string functions or do bound checking when possible.

Technique b) is potentially problematic if the developer cannot anticipate the input size accurately. The buffer size after increasing may still be not enough for an untested input in the future. As shown in Figure 2.9, the first fix was incomplete. After it increased the size of *avail* to 20, *avail* was still overflown

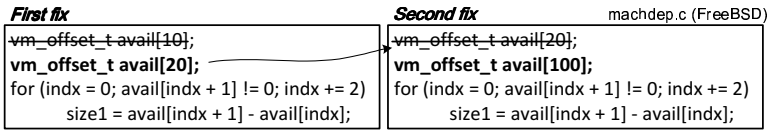

Figure 2.9: **Incorrect fix to a buffer overflow by increasing static buffer size.** The first fix enlarged the buffer size to 20, but the size was still not big enough. Under certain input, *avail* was still overflown.

later. Actually even the second fix might still be flawed. Since the developer does not add a bound check, a future input beyond the size 100 could still overflow *avail*.

Implications: Increasing the static buffer size can be dangerous if the input size cannot be accurately estimated.

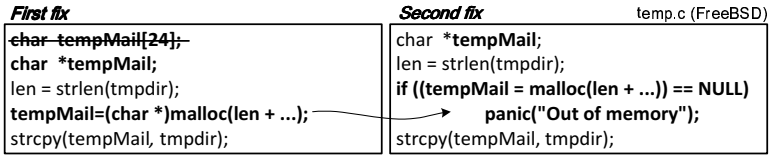

Figure 2.10: **Incorrect fix to a buffer overflow by allocating heap memory.** The first fix allocated heap memory to replace stack buffer, but the return value of malloc was unchecked.

For technique c), developers need to be aware of the rules to use memory allocation functions. The memory allocated needs to be freed after use, otherwise it may introduce a memory leak. Besides, developers need to consider handling the case if memory allocation fails. As shown in Figure 2.10, the fixer did fix the buffer overflow, but introduced a potential invalid memory access. It replaced the stack buffer (*tempMail*) with a dynamically allocated buffer, but forgot to handle the case if *malloc* fails.

Implications: When allocating memory dynamically to fix buffer overflow, developers also need to follow the safety rules of using memory allocation functions.

2.6.4 Fixing Memory Leaks

Once a memory leak is detected, writing fixes may be straightforward, but mistakes can still be made.

Specifically, fixing memory leak can introduce dangling pointer or null pointer dereference if the pointer would still be accessed after the free. Figure 2.11 shows an example where a dangling pointer bug was introduced. After p is freed, it can still be dereferenced (after testing against NULL) out of the function. Hence, the second fix further nullifies the pointer before the function return.

Implications: It is a good practice to nullify the pointer after freeing it, which can avoid dangling pointer bugs.

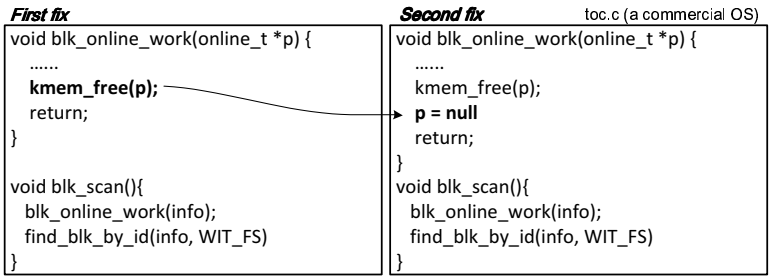

Figure 2.11: **Incorrect fix to memory leak introduced a dangling pointer.** The pointer p was later used in function *find_blk_by_id* with null pointer check. However, the first fix simply freed p without nullifying it.

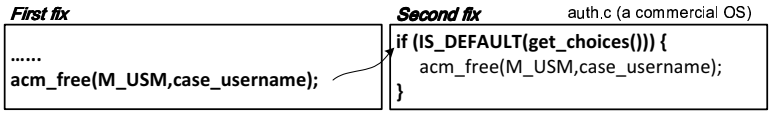

Figure 2.12: **Incorrect fix to a memory leak introduced data corruption.** The first fix freed the data indexed by *case_username* unconditionally. However, the data should be freed only under certain conditions.

Developer may also not be aware of the condition to free an object. They should only free an object when it is no longer used, not on those paths that it is still in use. If they overreact, they could mistakenly free an object still in use under certain conditions. Figure 2.12 shows such an example which led to data corruption.

Implications: Before fixing memory leak, developers should make sure when and what should be freed to avoid overreaction.

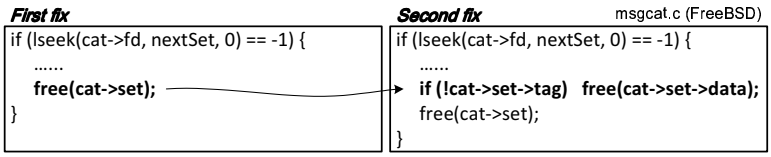

Figure 2.13: **Incomplete fix to a memory leak.** The first fix only freed *cat->set* but forgot to free its member *data*.

Besides, fixing memory leak can be incomplete. For some complex data structures, fixers may forget to free all their members. Figure 2.13 shows such an example.

Implications: For complex data structures, fixers should remember to free all their members.

2.6.5 Fixing Semantic Bugs

Semantic bugs have very diverse root causes, so the ways to fix them are also diversified. However, we still observed one common incorrect fix pattern for semantic bugs: conditions (e.g., *if* condition) are difficult to fix correctly. As shown in Figure 2.3, the first fix to the *if* condition was still not restrictive enough. Though this pattern is frequently observed, it is not easy to leverage current techniques to detect them. We think fixing semantic bugs correctly may require more application specific knowledge from fixers.

2.6.6 General Approaches to Detect Incorrect Fixes

Understanding the impact of the change: A fundamental reason for developers to make mistakes during bug fixing is that they do not know all the potential impacts of the newly fixed code. For example, in Figure 2.5, the fixer was not aware that the newly added lock *sc* would deadlock with the function *bus_teardown_intr*. In Figure 2.1, the developer is not aware that *buf* will need to store an out-of-band string. If all such potential "influenced code" (either through control- or data-dependency) is clearly presented to developers, they may have better chances to detect the errors. More specially, a tool that can show which variables are affected, which def-use chains are modified, which paths are deleted, added or altered, and which preconditions and postconditions are altered (e.g. a function will be called with a lock held now) are highly expected. We envision compiler techniques such as program slicing [102, 42, 107] that take the dependencies to the patch as the slicing criterion can be extended to analyze such information and used to build such tools.

Apply checkers incrementally As discussed before, it is possible for some existing bug detection tools (checkers) [58, 35, 36] to detect some types of incorrect fixes. However, applying these tools directly on the full code base after the fix is not practical. It may take a very long time for them to scan the entire code base, which may be redundant with the original testing steps, or not always necessary. Also it may produce too many false positives. Instead, developers may want to check the code influenced by the patch first. One observation is that sometimes, just checking within the function boundary is enough to detect problems in the patch. For example, as shown in Figure 2.2, a path-sensitive checker that simply checks the rule "lock is always paired with an unlock" can easily detect the missed *SOCK_UNLOCK* by only scanning the function that the patch modified. Though sometimes we need inter-procedural analysis to attain more information, in many cases the current detection tools can be simplified to just check the changes in patches. Then these tools can be very light-weighted so that developers would be more willing to using them.

Dealing with incomplete fixes Some incomplete fixes are introduced by the fact that fixers may forget

to fix all the buggy regions with the same root cause. This types of incomplete fixes can be mitigated by using technique [77, 72] which searches for other places that have the same patterns or usage scenarios in entire code. For example, in Figure 2.6, the first fix that suggested certain shared objects need to be protected. Then developers can try to find the other places where those objects are accessed without protection. However, this technique is less effective when a consistent pattern is difficult to learn. For instance, those incomplete fixes related to conditions (Figure 2.3 in Chapter 2.1) can not be solved by this technique. For these bugs, we think it is better to have more knowledge developers and reviewers in the loop so that incorrect fixes could be avoided. We will discuss this issue in Chapter 2.7.

2.7 Lack of Knowledge

In this section we study if programmers' code knowledge is a factor to incorrect fixes. The correctness of a fix will be influenced by both fixer and reviewer. Therefore, we study the knowledge from both the fixers and reviewers' perspective.

2.7.1 Fixer's Knowledge

Intuitively, if a file is totally written by a developer recently, the developer might have fresh and complete knowledge on the file. Then there will be less problem if the developer is the one who fixes the bug in this file. On the contrary, if a developer knows nothing about a file and he needs to fix a problem within this file, there might be higher probability that he could introduce an incorrect fix.

System	**Actual Fixer for Incorrect Fixes**		**Actual Fixer for Correct Fixes**		**Potential Optimal Fixer**	
	K_File	K_Func	K_File	K_Func	K_File	K_Func
A	13.2% (*0.022*)	18.1% (*0.046*)	18.3% (*0.019*)	20.5% (*0.012*)	65.0% (*0.043*)	75.1% (*0.031*)
B	9.5% (*0.013*)	11.5% (*0.023*)	15.4% (*0.016*)	27.9% (*0.031*)	39.7% (*0.024*)	51.4% (*0.022*)
C	12.8% (*0.024*)	16.1% (*0.037*)	17.2% (*0.021*)	18.4% (*0.023*)	69.8% (*0.031*)	78.1% (*0.026*)
D	7.9% (*0.017*)	12.5% (*0.035*)	15.5% (*0.024*)	16.4% (*0.021*)	78.0% (*0.023*)	78.4% (*0.039*)
AVG	10.9%	14.6%	16.6%	20.8%	63.1%	70.8%

Table 2.8: **The fixers' average code knowledge on the buggy files/functions.** The *variance* of the code knowledge is shown in the parentheses. Code knowledge is shown in the form of percentage (e.g., 13.2% means a knowledge value of 0.132). "Potential optimal fixer" is the developer with the most knowledge on the buggy files/functions but might not be always assigned the bug fixing task.

We first measured the K_File and K_Func (defined in Chapter 2.3.4) for the fixers who made the incorrect fixes. The results are shown in Table 2.8. We found that in general these fixers who made the incorrect fixes were not knowledgeable about the buggy files/functions. Specifically, they had only contributed on average 10.9% to the files and 14.6% to the functions involved in the patch before they made

the incorrect fix. In comparison, Table 2.8 also shows the fixers' knowledge in *correct* fixes. Selecting correct fix set is also challenging, since to verify the correctness of a fix is not trivial. Some fix might be "correct" now but would be found incorrect later. Here we just assume the fixes in *sample set 1* which are not classified as incorrect fixes are correct fixes. We found those fixers who made the correct fixes had contributed on average 16.6% to the files and 20.8% to the functions. In other words, the knowledge of the fixers who made the correct fixes is 1.5 times of that of the fixers who made the incorrect fix based on our code knowledge metrics, indicating source code knowledge could be a factor to incorrect fixes. We observed the knowledge of the fixers who made the correct fixes is also not significantly high. We suspect this is due to that the overall bug-report assignment process is not organized enough so that the developer who is the most knowledgeable about the code is usually not selected as a fixer.

But can we really find a developer who is *more knowledgeable* than the actual fixer of the incorrect fix? Table 2.8 also answered this question: surprisingly, by selecting the most knowledgeable developer who is still active in the development when the bugs need to be fixed as the fixer, the K_File and K_Func can reach as high as 63.1% and 70.8% respectively, which is 5∼6 times of the knowledge of the actual fixers in incorrect fixes. It should be emphasized that these "potential optimal fixers" are still reachable when the bugs were opened. We decide this by examining the check-in history of each developer. Our result suggests that the current bug fixing and reviewing process is not always assigning the problem to the developers who could be most "knowledgeable" with the bug.

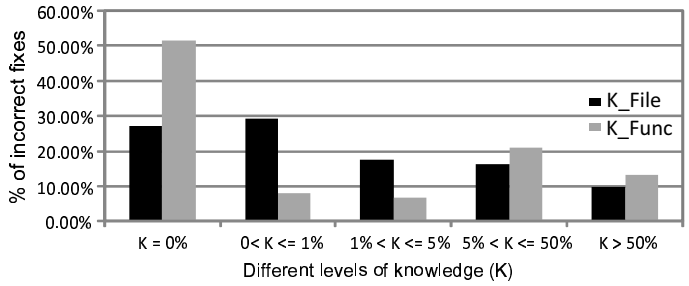

Figure 2.14: **The distribution of incorrect fixes in different knowledge scales.**

Figure 2.14 further studies why the K_File and K_Func in incorrect fixes are low by zooming into the distribution of incorrect fixes in different knowledge scales. We found the low K_File and K_Func are caused by a large portion of incorrect fixes that were made by fixers with *zero* prior knowledge to the buggy files/functions. As shown in Figure 2.14, *27.2% of the fixers had not contributed any lines to the file ($K\_File = 0\%$) they were about to fix.* It is even worse at the function level. 51.4% of fixers had not contributed any lines of code to the function they were about to fix. These "*first touches*" could be dangerous

since the developer could have little knowledge about the particular part of code when they are about to fix.

Besides studying the effect of code knowledge in terms of K_File and K_Func, we further studied the code knowledge in terms of whether the fixer actually fixed the the code previously written by him. The intuition behind this is that even though a fixer had written small amount of code (i.e., small K_File/K_func), as long as the fixer was modifying the code regions that he had written, he might be still considered as the knowledgeable person for the fix. Specifically, for each fix, if any code line modified by the fix was also previously written by the same fixer, we count this fixer is fixing his own code. In other words, we count the fixer is not fixing his own code only when none of the lines of code involved in the patch was written by him. The result is shown in Table 2.9.

System	For Incorrect Fixes	For Correct Fixes
A	7.7%	26.2%
B	16.4%	44.8%
C	18.2%	25.9%
D	8.0%	24.1%
AVG	12.6%	30.3%

Table 2.9: **The percentage of fixes that fixer is fixing his own code.** For example, the number 7.7% in the first cell means: among all the incorrect fixes from OS A, 7.7% of them were actually fixed by developers who were fixing their own code.

Table 2.9 shows large difference between correct fixes and incorrect fixes. The ratio of the fixers who fixed their own code for correct fixes is 2.5 times of the ratio for incorrect fixes (30.3% v.s. 12.6%). This suggests fewer fixers (in term of ratio) are fixing their own code for incorrect fixes than for correct fixes, which further suggests that fixing code written by others might be prone to incorrect fixes.

Moreover, knowledge may decay as time goes by. After a developer hasn't been working on a file for quite long, he may become unfamiliar with that file even if he had contributed a lot code to this file historically. To measure this perspective, we further did an experiment which tried to locate when was the developer's last modification on the file before the developer modified a file to fix the bug, We calculated the time gap between two modification points to analyze the knowledge decay. The results are shown in Table 2.10. For 73.5%$\sim$86.2% of developers, their last modification on the file before their incorrect fix is one month ago. For 32.6%$\sim$58.0% of developers, this gap is even half year ago.

System	More Than 1 Week	More Than 1 Month	More Than 3 Months	More Than Half Year
A	77.2%	73.5%	50.4%	32.6%
B	93.0%	74.8%	62.9%	48.2%
C	85.4%	78.1%	65.7%	58.0%
D	91.7%	87.2%	57.7%	43.3%

Table 2.10: **The gap between the last modification and incorrect fix.**

Inspired by our study in code knowledge, a software vendor is building a tool to find knowledgeable fixers and reviewers. Sometimes when the bug report just arrives it may be unclear which files/functions contain the bug. In these cases the knowledge is more useful to assign *reviewers* who are familiar with the files/functions involved in the fix after the fix is made. This knowledge can also be used in prioritizing patch testing efforts to pay more attentions to the patches fixed by less-knowledgeable fixers.

Finding: The fixers' knowledge on the buggy files/functions in correct fixes is 1.5 times of the fixers' knowledge in incorrect fixes. Fewer fixers (in term of ratio) are fixing their own code for incorrect fixes than for correct fixes. Moreover, nearly 27% of the incorrect fixes are made by developers who have not contributed a single line to the entire file they are about to fix. The potential "Optimal" developer who is most familiar with the buggy code has 5∼6 times knowledge of that of the actual fixer in incorrect fixes.

Implication: It might be beneficial to assign the bugs to developers with more knowledge during the bug-triage process. The knowledge can also be considered as a factor in prioritizing the testing efforts on patches.

2.7.2 Reviewer's Knowledge

Once the patch is made by the developer, the patch usually needs to pass a reviewing process before releasing to customers. Since at this time, the information about which files and functions are modified is already available, it is possible for us to select an optimal set of reviewers who have the best knowledge on that part of code. And when those knowledgeable reviewers are on duty, it might be easier for them to detect the problems hidden in the patch. Therefore we also measured K_File and K_Func for the reviewers on both the incorrect fixes and correct fixes to test our assumption. The results are also shown in Table 2.11.

System	Reviewer for Incorrect Fixes		Reviewer for Correct Fixes		Optimal Reviewers	
	K_File	K_Func	K_File	K_Func	K_File	K_Func
A	19.7%	18.7%	19.7%	15.6%.	80.0%	89.1%
B	9.0%	11.2%	11.8%	13.5%	49.4%	64.4%
C	13.1%	12.7%	7.6%	13.4%	75.3%	82.9%
D	N/A	N/A	N/A	N/A	N/A	N/A
AVG	13.9%	14.2%	12.8%	14.1%	68.2%	78.8%

Table 2.11: **The reviewers' average knowledge to the buggy code.** "Optimal Reviewers" are the *top two* developers with the most knowledge to the buggy code, since usually there are two reviewers for each patch. For the code base D, we cannot find the information about reviewers from its bug database and SCM, so the entries are marked with "N/A".

Most of the reviewers who reviewed the incorrect fixes also had little knowledge on the files and functions they reviewed. Comparing to the reviewers on correct fixes, the knowledge for reviewers on correct fixes is sometimes lower (for A) than the knowledge for reviewers on incorrect fixes, sometime higher (for B) and sometimes mixed (for C). The results don't suggest reviewers' knowledge don't have influence on the correctness of fixes. Still we believe knowledgeable reviewers will be more helpful. We looked into the set of correct fixes. We found for correct fixes where fixers have low knowledge (less than 10%), the reviewers' knowledge is usually higher than the reviewers' average knowledge, which suggests the benefits of having knowledgeable reviewers.

To demonstrate the potential improvement we can achieve, we also did an enhanced assignment strategy which always try to select the most knowledgeable developers on that part of code when those people are available. Our results are still shown in Table 2.11 under the "Optimal Reviewer" columns. It shows a 6x-7x improvement on the knowledge of reviewers. Though future user study need to be conducted to evaluate the direct impact on the incorrect fix due to this knowledge improvement on reviewers, we think it is definitely good to adopt such changes.

Finding: The reviewers who review the incorrect fixes also might not have enough knowledge on the related code. However, there were developers with better knowledge on the code who were not assigned as reviewers.
Implication: The reviewing process could be potentially improved.

2.7.3 Towards Building a Predictor

In previous sections, we had discovered that the code knowledge of fixers and reviewers could influence the chance of incorrect fixes. Then it is good to leverage such information to predict the probability that a fix could be incorrect. Here we want to emphasize that to predict is not to detect. A detector can usually pinpoint where the exact error is within a fix. A predictor can only tell if a fix is likely to be incorrect. Developers need to further look at the fix to verify the prediction. However, the process of bug fixing can still benefit from a good incorrect fix predictor. For software vendors, especially large vendors, there are always many bugs that they have fixed. However, the limited testing resources prevent them from carefully examining every bug fix. A good predictor can prioritize the code reviewing to those fixes that are more likely to be buggy.

To build a predictor, we need first select the features that could have relation with incorrect fixes. It is hard to have some prior knowledge on which features are most predictive before running any experiment, so we first select seven features based on our domain knowledge. Another reason we chose these features is

that they are relatively easy to measure. Some features, though might be related, are hard to measure or not documented. Therefore we did not include them in our selection. We do not claim this selection to be exhaustive. New features can be further added to refine the predictor.

The features we selected are:

kfi_t1_f if the fixer is the most knowledgeable developer (top1) of the file.

kfi_t3_f if the fixer is among the three most knowledgeable developers (top3) of the file.

kfi_t1_r if any reviewer is the most knowledgeable developer (top1) of the file.

kfi_t1_r if any reviewer is among the most knowledgeable developers (top3) of the file.

$last\_modify$ how long ago it is the fixer's last modification to the file. If the fixer has never touched the file before, we used a default maximum value (e.g., 10 years).

num_of_author how many authors have contributed to the file.

loc the number of code lines in the file.

Here the first five features are all more or less related to the code knowledge. num_of_author can also influence code knowledge in an implicit way. Since if there are too many co-authors for a file, each author is less likely to be "very" knowledgeable about the file. *loc* is a widely used feature among various defect predictors [91, 52]. Including it may give us a reference on the predictive power of those knowledge related features.

We used Weka [15] to do the feature selection. We chose information gain as the evaluator and ranker as the search scheme. Besides, we also used a ten fold cross validation. Each fix can be represented as a vector of these seven features and is labeled as either "correct fix" or "incorrect fix". For convenience, we ran the experiment on the data from the commercial company and the results are shown in Table 2.12.

Feature	Average Merit
kfi_t1_f	0.213±0.032
kfi_t3_f	0.240±0.025
kfi_t1_r	0.141±0.048
kfi_t3_r	0.157±0.051
$last\_modify$	0.125±0.039
num_of_author	0.263±0.034
loc	0.128±0.041

Table 2.12: **The average merit for the eight features when they are used to predict whether a fix is correct or not.** Here for convenience, we just used the data from the commercial company.

As shown in the Table 2.12, num_of_author actually has the highest predictive power, followed closely by kfi_t3_f. It may be a little bit surprising that num_of_author ranks such top. We guess this suggested that it might not be good to have too many authors for a file. If that happens, it not only increases the complexity of management and communication, but also suppresses the average code knowledge. kfi_t3_f is more predictive than kfi_t1_f. This suggests that it will be too restrictive if we only look at if a fixer is the top1 knowledgeable developer. In order to achieve a good discrimination, we should relax the constraint a little. We also found generally the code knowledge of fixers are more important than that of reviewers. We think that this is because fixer is the direct factor that decides the correctness of a fix. Besides, reviewers may not spend all their efforts in reviewing the fix and some error in the fix might be hard to detect. This suggests that it might be more beneficial to find a knowledgeable fixer. *loc* and $last\_modify$ are the two least predictive ones. Actually *loc* is frequently used in a lot of defect predictors, but our experiment suggested that it might not be a good feature to predict incorrect fix.

Our experiment only shows the prospect of building a predictor for incorrect fixes including the features that relate to code knowledge. To build a more solid and reliable predictor will remain as our future work.

Chapter 3

The Characteristic Study on Misconfigurations

3.1 Overview

To study misconfigurations, first we need to fully understand what a misconfiguration is. Most previous work has just focused on errors in configuration parameters (i.e., entries in configuration files). Do misconfigurations only refer to configuration parameters? According to Federal Standard 1037C [16], "configuration" is defined as: *"In communications or computer system, an arrangement of functional units according to their nature, number, and chief characteristics. Configuration pertains to hardware, software, firmware, and documentation."* Therefore, besides configuration parameter errors, misconfigurations also include other types.

In our study, we not only studied configuration parameters, but also looked into other types of configurations, such as how a system is organized and whether different modules are compatible with each other. Besides, we also briefly studied the miconfigurations that happen at hardware layer.

There are many attributes (questions) related to a configuration problem. We should choose those that can give us enough insights on solving the problem of misconfigurations, or even avoid the occurrence of misconfigurations fundamentally.

We think it would be useful to know the common root causes, mistake patterns, and impacts of misconfigurations. For example, how are misconfigurations introduced? What fraction of them are introduced due to software upgrades? For parameter-based misconfigurations, how many parameters are usually involved? What percentages of misconfigurations involve multiple machines or components? What percentages of misconfigurations violate the syntax or format requirements of configuration files, and what percentage of misconfigurations have perfectly legal parameter values and are simply not delivering the functionality intended by users? What kind of configurations are error-prone? What are the impacts of misconfigurations? Do they make systems fully or partially unavailable? The answers to the above questions are useful for building more effective tools to detect, diagnose or fix misconfigurations automatically.

Moreover, it would also be helpful to know how today's systems react to misconfigurations. Do they

Findings on High Level Questions (Chapter 3.4)	
Are configuration issues prevalent?	Similar to the results from previous studies [43, 79, 75], our study using data from a commercial vendor also shows a significant portion (27%) of customer cases are related to configuration issues.
Are configuration issues severe?	Configuration issues are not of low severity as is commonly thought. In fact, their percentage is increased to 31% if we consider only high severity problems reported by customers.
Findings on Misconfiguration Types (Chapter 3.5)	
What types of misconfigurations are there?	Configuration parameter mistakes account for the majority (70.0%∼85.5%) of the examined misconfigurations.
	However, a significant portion (14.5%∼30.0%) of the examined misconfigurations are caused by software compatibility and component configuration, which are not well addressed so far.
What types of parameter mistakes are there?	38.1%∼53.7% of parameter misconfigurations are caused by illegal parameters that clearly violate some format or semantic rules defined by the system, and can be potentially detected by checkers that inspect against these rules.
	However, a large portion (46.3%∼61.9%) of the examined parameter misconfigurations have legal parameters but do not deliver the functionality intended by users. These cases are more difficult to detect by automatic checkers and may require more user training or better configuration design.
Any special pattern?	A significant portion (12.2%∼29.7%) of parameter mistakes are due to value-based inconsistency, calling for an inconsistency checker or a better configuration design that does not require users to worry about such error-prone consistency constraints.
Is a misconfiguration always within the application of interest?	Although most misconfigurations are located within each examined application, a significant portion (21.7%∼57.3%) involve configurations beyond the application itself or span over multiple hosts.
System Reaction to Misconfigurations (Chapter 3.6)	
Do systems detect the misconfig?	Only 7.2%∼15.5% of the studied misconfiguration problems provide explicit messages pinpoint the location of the error.
Mysterious reaction?	Misconfiguration cases can make systems to misbehave such as crash, hanging or severe performance degradation, making failure diagnosis a challenging task.
Possible to have better msgs?	Among the 208 cases with illegal parameters, fewer than 20% of them provide explicit messages. Up to 31.3% of them did not provide any message at all.
Benefit of explicit msgs.	Explicit messages that pinpoint configuration errors can shorten the diagnosis time as much as 14.5 times comparing to the cases with in-explicit messages or no messages at all.
Findings on Causes of Misconfigurations (Chapter 3.7)	
Did the system use to work?	For simpler systems, the majority of misconfigurations are related to first-time use of certain desired functionality. For more complex and larger systems, a significant percentage (16.7%∼32.4%) of the misconfigurations were introduced into systems that used to work fine.
Why did it work before, but now doesn't?	By looking into the 100 "used-to-work" cases (32.4% of the total) at COMP-A, 46% of them are attributed to configuration parameter changes due to routine maintenance, configuring for new functionality, system outages, etc, and can benefit from tracking configuration changes. The remainder are caused by non-parameter related issues such as hardware changes (18%), external environmental changes (8%), resource exhaustion (14%), and software upgrades(14%).
Findings on Impact of Misconfigurations (Chapter 3.8)	
How much impact do misconfigurations have?	Although most studied misconfiguration cases only lead to partial unavailability of the system, 16.1%∼47.3% of them make the systems to be fully unavailable or cause severe performance degradation.

Table 3.1: **Our major findings of real world misconfiguration characteristics.** Please take our methodology and potential threats to validity into consideration when you interpret and draw any conclusions.

detect misconfigurations and have explicit messages pointing out the configuration problem, or are they not well prepared for misconfigurations, or even worse, do they result in crashes, hangs or other types of failures that are very hard to diagnose and differentiate from other root causes (e.g., software bugs or hardware errors)? Some analysis of existing systems would help guide developers in implementing systems that are better prepared for misconfigurations and reduce the amount of time spent by support engineers in troubleshooting failures caused by configuration errors.

Finally, a misconfiguration characteristic study can also help developers design better configuration logic that is less prone to configuration errors. Misconfigurations are different from software bugs. The latter are introduced solely by developers; the former lie in a gray zone between developers and users because either party could be responsible for a misconfiguration. This undefined responsibility may confuse developers when they are designing the configuration logic, constraints and requirements of a system. Therefore, some "vulnerabilities" can be introduced into the configuration requirements which make users prone to configuration mistakes. Understanding the major types and root causes of misconfigurations may help guide developers to better design configuration logic and requirements, and reduce the likelihood of configuration mistakes by users.

All the above motivations are calling for a comprehensive characteristic study on real world misconfigurations. Unfortunately, in comparison to software bugs that have well-maintained bug databases and have benefited from many software bug characteristic studies [94, 95, 33, 60], a misconfiguration characteristic study is much harder, mainly because historical misconfigurations usually have not been recorded rigorously in databases.

In this thesis, we sampled over 3,000 custom reported cases from both commercial system (COMP-A's system and open source systems (CentOS, MySQL, Apache and OpenLDAP). From the over 3,000 sampled cases, we manually identified 546 misconfiguration cases and studied their characteristics based on the questions we raised previously. Among these 546 cases, 309 cases are from the commercial system and the other 237 cases are from the four open source systems. Our major findings are listed in Table 3.1.

While we believe that the misconfiguration cases we examined well represent the characteristics in large system software, we do not intend to draw any general conclusions about all the applications. In particular, we should note that all of the characteristics and findings in this study are associated with the types of the systems. Therefore, our results should be taken with the specific system types and our methodology in mind (discussed in Chapter 3.3).

3.2 Background

Characteristic studies on operator errors: Several previous works have examined the contribution of operation errors or administrator mistakes [43, 69, 79, 75, 71, 73]. For example, Jim Gray found 42% of system failures are due to administration errors [43]. Patterson et al. [79] also observed a similar trend in telephone networks. Murphy et al. [69] found the percentage of failures due to system management is increasing over time. Oppenheimer et al. [75] studied the failures of the Internet services and found that configuration errors are the largest category of operator errors. Nagaraja et al. [71] also had similar findings from a user study.

To the best of our knowledge, very few studies have dived deeper into misconfigurations and examined the subtypes, root causes, impacts and system reactions to misconfigurations, especially in both *commercial* and open source systems with a large set of real world misconfigurations.

Detection of misconfigurations: A series of work [99, 101, 38, 17] in recent years tried to detect misconfigurations. The techniques used in Peerpressure [99] and its predecessor Strider [101] have been discussed in Chapter 1.2. RCC [38] checks the router configuration against an abstract model of BGP protocol to detect the misconfiguration mistakes. Microsoft Baseline Security Analyzer (MBSA) [17] detects common *security-related* misconfigurations by checking configuration files against predefined rules.

Diagnosis of misconfigurations: Besides detection, another series of work [103, 92, 24, 25] tried to diagnose problems after the errors happen. We have already discussed AutoBash [92], ConfAid [25], and Chronus [103] in Chapter 1.2. The applicability of Chronus depends on how many misconfigurations belong to the "used-to-work" category. A follow-up work to AutoBash by Attariyan et al. [24] leverages system call information to track the causality relation, which overcomes the limitations of the Hamming distance comparison used in AutoBash to further enhance accuracy. Similar to [24], Yuan et al. [106] uses machine learning techniques to correlate system call information to problem causes in order to diagnose configuration errors. Most of these works focused on parameter-related misconfigurations.

Tolerance of misconfigurations: Some research work [92, 30] can help fix or tolerate misconfigurations. In addition to AutoBash [92] (as discussed in Chapter 1.2), Undo [30] uses checkpoints to allow administrators to have a chance to roll back if they made some misconfigurations. Obviously it assumes that the system used to work fine, which is true for some cases but not all as our results have shown.

Avoidance of misconfiguration: One approach to avoid misconfiguration is to develop tools to configure the system automatically. SmartFrog [19] uses a declarative language to describe software components, configuration parameters and how they should connect to each other. Configurations can then be automat-

ically generated to greatly mitigate human errors. Similarly, Zheng et al. [110] leverage custom specified templates to automatically generate the correct configuration for a system. Kardo [54] adopts machine learning techniques to automatically extract the solution operations out of user's UI sequence and apply them automatically. These solutions then can be used to help users to configure their system automatically.

A more fundamental approach is to design better configuration logic/interface to avoid misconfigurations at the first place. Maxion et al. [62] discovered that many misconfigurations for NTFS permission are due to that the configuration interfaces do not provide adequate information to users. Therefore, they proposed a new design of interface with subgoal support which can effectively reduce the configuration errors on NTFS permission by 94%.

Misconfiguration injection: As mentioned in Chapter 1.2, a misconfiguration injection framework like Conferr [50] is very useful in evaluating techniques for detecting, diagnosing and fixing misconfigurations. Our study can be beneficial for such framework to construct a more accurate misconfiguration model.

Online validation: Another avenue of work [71, 73, 74, 34] tried to validate the system to detect operator mistakes. Nagaraja et al. [71] developed a validation framework which can detect operator mistakes before deployment by comparing against the comparator functions provided by users. A follow-up work by Oliveira et al. [73] tried to validate database system administrations. Another follow-up work by Oliveira et al. [74] tried to address the limitation of previous validation system which does not protect against human errors directly performed on the production system. Mirage [34] also has a sub-system for validating the upgrading of the system.

Misc.: Wang et al. [100] used reverse engineering to extract the security related configuration parameter automatically. People can use this to slice the configuration file and to see if the security related parameters are correct. Ramachandran et al. [85] tried to extract the correlations between parameters, which can be used to detect some of the inconsistency misconfigurations in our study. Rabkin et al. [83] found that the configuration space after canonicalization is not very big after having analyzed seven open-source applications. Therefore a thorough test of different configuration parameters might be possible for certain applications if input is generated in a smart way.

As we discussed in Chapter 1.2, our characteristic study of real world misconfigurations would be useful to provide some guidelines to evaluate, improve and extend some of the above work on detecting, diagnosing, and fixing misconfigurations, as well as misconfiguration injection.

3.3 Methodology

There are unique challenges in obtaining and analyzing a large set of real-world misconfigurations. Historically, unlike bugs that usually have Bugzillas as repositories, misconfigurations are not recorded rigorously. Much of the information is in the form of unstructured textual descriptions and there is no systematic way to report misconfiguration cases. Therefore, As one of the first attempts to study misconfiguration characteristics, our study is a best-effort one. In order to overcome these challenges, we manually analyzed reported misconfiguration cases by studying manuals, instructions, source code, and knowledge bases of each system. For some hard cases, we contacted the corresponding engineers through emails or phone calls to understand them thoroughly. The overall effort took about 21 man-months, excluding the help from several COMP-A engineers and open source developers.

3.3.1 Data Sets

We examined misconfiguration data for one commercial system and four open-source systems. The commercial system is the storage system from COMP-A, one of the primary storage vendors in the world. The core software running in such system is proprietary to COMP-A, including its operating system that manages the storage system. The open source systems include CentOS, MySQL, Apache HTTP server, and OpenLDAP. We focused primarily on software misconfigurations. We will only briefly talk about hardware misconfigurations in Chapter 3.5.7, since there is no sufficient data for hardware misconfigurations on systems running the open-source software.

COMP-A storage systems consist of multiple components including storage controllers, disk shelves, and interconnections between them (e.g., switches). These systems can be configured in a variety of ways for customers with different degrees of expertise. For instance, COMP-A offers tools that greatly simplify system configuration (e.g., Provisioning Manager [81], Protection Manager [9], etc.) and tools that perform automatic configuration analysis to detect misconfigurations and identify potential compatibility issues for software upgrades [97]. We cannot ascertain from the data whether users configured the systems directly or used tools for configuration.

The misconfiguration cases we studied are from COMP-A's official customer-issues database, which records problems reported by customers. For accuracy, we considered only closed cases, i.e. cases that COMP-A has provided a solution to the users. Also, to be as relevant as possible, we focused on only cases over the last two years. COMP-A's support process is rigorous, especially in comparison to open-source projects. For example, when a customer case is closed, the support engineer needs to record information about the root cause as well as resolution. Such information is very valuable for our study. There are many

cases labeled as "Configuration-related" by support engineers and it is prohibitively difficult to study all of them. Therefore, we randomly sampled 1,000 cases labeled as related to configuration. Not all 1,000 cases are misconfigurations because more than half of them are simply customer questions related to how the system should be configured. Hence we did not consider them as misconfigurations. We also pruned out a few cases for which we cannot determine whether a configuration error occurred. After careful manual examination, we identified 309 cases as misconfigurations, as shown in Table 3.2.

Besides COMP-A storage systems, we also studied four open-source systems: CentOS, MySQL, Apache HTTP server, and OpenLDAP. All are mature software systems, well-maintained and widely used. CentOS is an enterprise-class Linux distribution, commonly used by organizations and individuals. We chose CentOS because it is a full-fledged operating system with various applications and services built on top. CentOS includes a multi-layer software stack, from the operating system to services and applications running on it. We also selected three other open source systems to represent widely used Internet services, including a database server (MySQL), a web server (Apache HTTP server) and a directory server (OpenLDAP).

For open source software, the misconfiguration cases come from three sources: official user-support forums, mailing lists, and ServerFault.com (a large question-answering website focusing on system administration). Whenever necessary, scripts were used to identify cases related to systems of interest, as well as to remove those that were not confirmed by users. We then randomly sampled from all the remaining candidate cases (the candidate set sizes and the sample set sizes are also shown in Table 3.2) and manually examined each case to check if it is a misconfiguration. Our manual examination yielded a total of 237 misconfiguration cases from these four open-source systems. The yield ratio (used cases/sampled cases) is low for these open-source projects because we observe a higher ratio of cases that are customer questions among the samples from open source projects as compared to the commercial data

System	Total Cases	Sampled Cases	Used Cases
COMP-A	confidential	1000	309
CentOS	4338	521	60
MySQL	3340	720	55
Apache	8513	616	60
OpenLDAP	1447	472	62
Total	N/A	3329	546

Table 3.2: **The systems we studied and the number of misconfiguration cases we identified for each of them.**

3.3.2 Threats to Validity and Limitations

Many characteristic studies suffer from limitations such as the systems or workloads not being representative of the entire population, the semantics of events such as failures differing across different systems, and so

on. Given that misconfiguration cases have considerably less information than ideal to work with, and that we need to perform all of the analysis manually, our study has a few more limitations. We believe that these limitations do not invalidate our results; at the same time, we urge the reader to focus on overall trends and not on precise numbers. We expect that most systems and processes for configuration errors would have similar limitations to the ones we face. Therefore, we hope that the limitations of our methodology would inspire techniques and processes that can be used to record misconfigurations more rigorously and in a format amenable to automated analysis.

Representativeness of system: We selected these software systems for two reasons: (1) they are mature and widely used, and (2) they have a large set of misconfiguration cases reported by users. While we cannot draw conclusions about any general system, our examined systems are representative of typical large, server-based systems.

Sampling: To make the time and effort manageable, we sampled the data sets. As shown on Table 3.2, our sample rates are statistically significant and our collections are also large enough to be statistically meaningful [40]. In our result tables, we also show the statistical errors with a 95% confidence level based on our sampling rates. we focus on software misconfigurations, and only briefly look into hardware misconfigurations since we do not have sufficient data on such cases from the open source systems that we examined.

Users: The sources from which we sample contain only user-reported cases. Users may choose not to report trivial misconfigurations. Also, it is more likely that novice users may report more misconfiguration problems. We do not have sufficient data to judge whether a user is new or an expert. But with new systems or major revisions of an existing system deployed to the field, there will always be new users. Thus in that sense our findings are still valid.

User environment: Some misconfigurations may have been prevented, or detected and resolved automatically by the system or other tools. This scenario is particularly true for COMP-A systems. At the same time, some, but not all, COMP-A customers use the tools provided by COMP-A and we cannot distinguish the two in the data.

System versions: We also observe that software is constantly evolving. We do not differentiate between system versions. It is possible that some of the reported configuration issues have already been or are being addressed during system evolution (e.g., automatically correcting it, providing better error messages, etc.). Therefore some of the reported configuration issues may not apply to some versions. However, since we studied the cases in the last 2 years, such cases should be rare in our data sets.

Overall, our study is representative of user-reported misconfigurations which are more challenging, urgent,

or important.

3.4 High Level Questions

Before diving into misconfigurations, we first look at how prevalent they are in the field and how severely they impact customers. For this, we analyzed all the customer reported cases over the last two years from COMP-A. There are five root causes classified by COMP-A engineers after resolving each customer reported problem: "configuration" (configuration related), "hardware failure", "bug", "customer environment" and "user knowledge". The first three types of root causes are easy to understand with their literal meaning. For the latter two, "customer Environment" majorly refers to the cases caused by power supplies, cooling systems or other environmental issues, while "user knowledge" primarily consists of the cases where customers just ask various technical questions without mentioning a particular failure. Each case is also labeled with a severity level by customer support engineers, ranged from "1" to "4", based on how severe the problem is in the field. Cases with severity level of "1" or "2" are usually considered as high severity cases that require prompt responses.

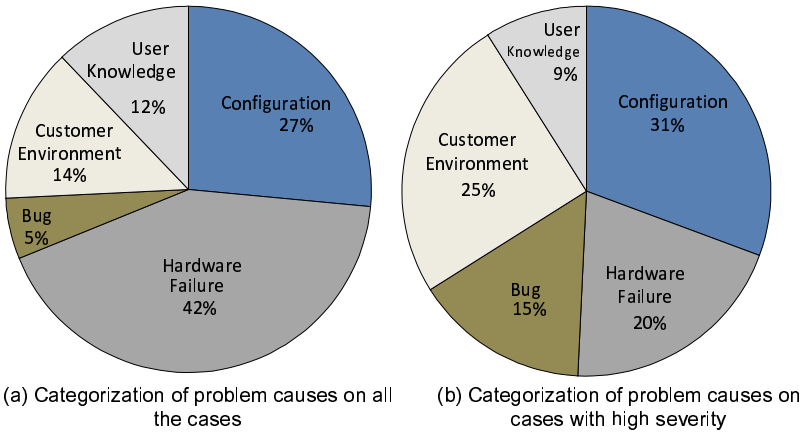

Figure 3.1: **Root cause distribution among the customer problems reported to COMP-A.**

Figure 3.1(a) shows the distribution of customer cases based on different root causes. Figure 3.1(b) further shows the distribution of *high severity* cases. We do not have the results for the open source systems due to unavailability of such labeled data (i.e., customer issues caused by hardware, software bugs, configurations, etc.).

Among all five categories, configuration related issues contribute to 27% of the customer reported cases and are the second most pervasive root cause of customer problems. Although hardware failures caused more customer reported problems, they have lower average priority than the problems caused by misconfiguration.

Furthermore, considering only high severity cases, configuration related issues become the most significant contributor. They contribute to 31% of high severity cases, which is ahead of 25% by customer environment, 20% by hardware failure and 15% by bugs. So we summarize the results into the following findings:

Finding: Similar to the results from previous studies [43, 79, 75], our study using data from a commercial vendor also shows a significant portion (27%) of customer cases are related to configuration issues.

Finding: Configuration issues are not of low severity as is commonly thought. In fact, their percentage is increased to 31% if we consider only high severity problems reported by customers.

3.5 Misconfiguration Types

3.5.1 Distribution among Different Types

To zoom into misconfigurations, we first looked at what types of misconfigurations are in the real world and their distributions. We classified the examined misconfiguration cases into three categories:

Parameter configuration parameter mistakes. A parameter could be either an entry in a configuration file or a console command for configuring certain functionality.

Compatibility misconfigurations related to software compatibility (i.e. whether different components or modules are compatible with each other).

Component other remaining software misconfigurations (e.g., a module is missing).

System	Parameter	Compatibility	Component	Total
COMP-A	246 (79.6±2.4%)	31 (10.0±1.8%)	32 (10.4±1.8%)	309
CentOS	42 (70.0±3.7%)	11 (18.3±3.1%)	7 (11.7±2.6%)	60
MySQL	47 (85.5±2.3%)	0	8 (14.5±2.3%)	55
Apache	50 (83.4±2.8%)	5 (8.3±2.1%)	5 (8.3±2.1%)	60
OpenLDAP	49 (79.0±3.0%)	7 (11.2±2.3%)	6 (9.7±2.2%)	62

Table 3.3: **The numbers of misconfigurations of each type**. Their percentages and the sampling errors are also shown.

Table 3.3 shows the following two findings:

Finding: Configuration parameter mistakes account for the majority (70.0%∼85.5%) of the examined misconfigurations.

Finding: However, a significant portion (14.5%∼30.0%) of the examined misconfigurations are caused by software compatibility and component configuration, which are not well addressed so far.

First, our findings support recent research efforts such as [99, 103, 92, 25] on detecting, diagnosing or fixing parameter-based misconfigurations. Second, the findings should send a signal to system designers to have fewer "knobs" (i.e. parameters) or fewer complicated "knobs" for users to configure and tune their system.. Whenever possible, auto-configuration [110] should be preferred because in many cases users may not be as sophisticated and experienced as the developers may expect.

Our findings also call for attention to investigating solutions dealing with non-parameter-based configurations such as software incompatibility, etc. For example, system designers or release engineers may need to think about shipping a complete package, or even using an appliance (whether virtual or physical) deployment model or a software-as-a-service (SaaS) model to reduce these incompatibility and general configuration issues.

3.5.2 Parameter Misconfigurations

In this section, we study the types of parameter mistakes (as shown in Table 3.4) and the number of parameters needed to be considered to diagnose or to fix a parameter misconfiguration.

Types of mistakes in parameter configuration First, we look at parameter mistakes that clearly violate some implicit or explicit configuration rules related to format, syntax or semantics. We call them "*illegal*" misconfigurations because they are unacceptable to the examined system. Figure 3.2(a)∼(g) show seven such examples. These types of misconfigurations may be detected automatically by checking against configuration rules.

System	Legal	Illegal					
		Format		Value			
		Lexical Mistakes	Syntax Mistakes	Typo	Value Inconsist with Values	Value Inconsist with Env	Others
COMP-A	114(46.3±6.1%)	10(4.1±2.4%)	5(2.0±1.7%)	3(1.2±1.3%)	73 (29.7±5.6%)	32(13.0±4.1%)	9(3.7±2.3%)
CentOS	26 (61.9±13.8%)	1(2.4±4.4%)	0	2(4.8±6.0%)	6 (14.3±10.0%)	6(14.3±10.0%)	1(2.4±6.0%)
MySQL	24(51.1±12.7%)	1(2.1±3.6%)	0	0	7(14.9%±9.0%)	8(17.0%±9.5%)	7(14.9±9.0%)
Apache	27(54.0±13.3%)	3(6.0±6.3%)	3(6.0±6.3%)	1(2.0±3.7%)	7(14.0±9.3%)	5(10.0±8.0%)	4(8.0±7.3%)
OpenLDAP	23(46.9±11.5%)	7(14.3±8.0%)	11(22.4±9.6%)	0	6(12.2±7.5%)	1(2.0±3.2%)	1(2.0±3.2%)

Table 3.4: **The distribution of different types of parameter mistakes for each application**.

In contrast, some other parameter mistakes are perfectly "*legal*", but they are incorrect simply because they do not deliver the functionality or performance desired by users, like the examples in Figure 3.2(h)

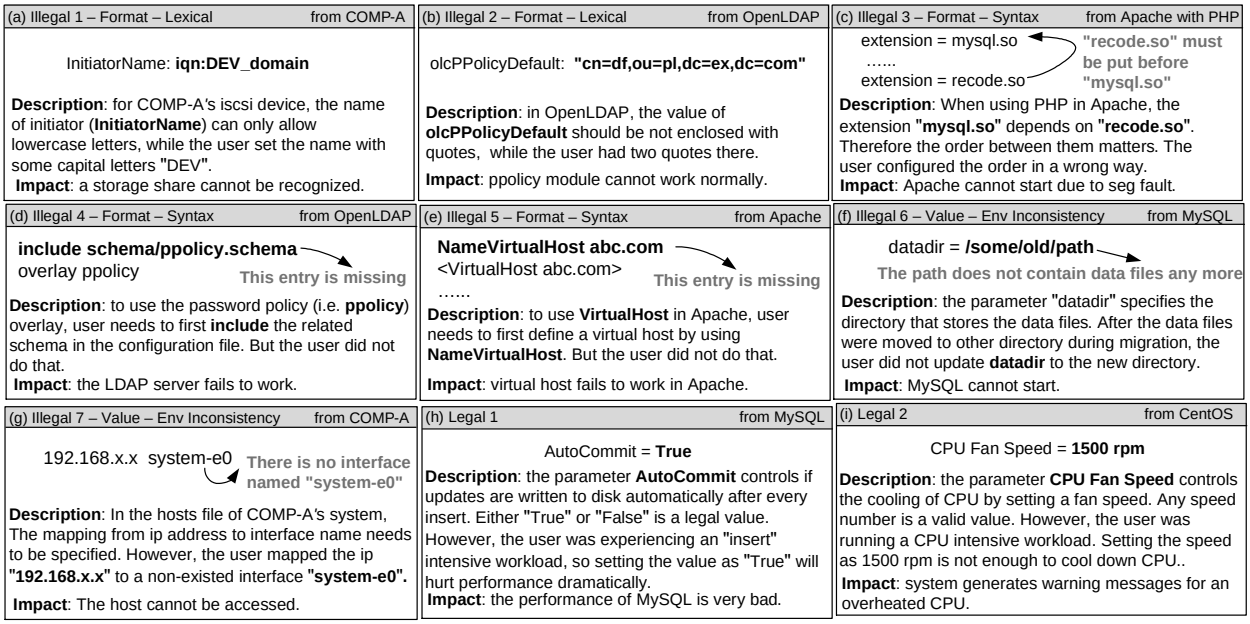

Figure 3.2: **Examples of different types of configuration parameter related mistakes.** ("legal" vs. "illegal", "lexical error", "syntax error" and some "inconsistency error".) The examples of the type "value inconsistency" will be shown in Chapter 3.5.3.

and (i). These kinds of mistakes are difficult to detect unless users' expectation and intent can be specified separately and checked against configuration settings. We think more user training may reduce these kinds of mistakes, as can simplified system configuration logic, especially for things that can be automatically configured by systems themselves.

> **Finding:** 38.1%∼53.7% of parameter misconfigurations are caused by illegal parameters that clearly violate some format or semantic rules defined by the system, and can be potentially detected by checkers that inspect against these rules.
>
> **Finding:** However, a large portion (46.3% ∼61.9%) of the parameter misconfigurations have perfectly legal parameters but do not deliver the functionality intended by users. These cases are more difficult to detect by automatic checkers and may require more user training or better configuration design.

To further look into illegal misconfigurations, we sub-categorize them into "*illegal format*", in which some parameters do not obey format rules such as lower case, field separators, etc.; and "*illegal value*", in which the parameter format is correct but the value violates some constraints, e.g., the value of a parameter should be smaller than some threshold.

Illegal format misconfigurations include both "*lexical*" and "*syntax*" mistakes. Similar to lexical and

syntax errors in program languages, a *lexical* mistake violates the grammar of a single parameter, like the examples shown in Figure 3.2(a) and (b); a *syntax* mistake violates structural constraints of the format, like the examples shown on Figure 3.2(c), (d) and (e). As shown in Table 3.4, up to 14.3% of the parameter misconfigurations are lexical mistakes, and up to 22.4% are syntax mistakes.

Illegal value misconfigurations mainly consist of "*value inconsistency*" and "*environment inconsistency*" mistakes. *Value inconsistency* means that some parameter settings violate some relationship constraints with some other parameters, while *environment inconsistency* means that some parameter's setting is inconsistent with the system environment (i.e., physical configuration). Figure 3.2(f) and (g) give two environment inconsistency example. As shown in Table 3.4, value inconsistency accounts for 12.2%∼29.7% of the parameter misconfigurations, while environment inconsistency contributes 2.0%∼17.0%. Both can be detected by some well designed checkers as long as the constraints are known and enforceable. We will further discuss this type of parameter mistakes in details in Chapter 3.5.3.

Finding: A significant portion (12.2%∼29.7%) of parameter mistakes are due to value-based inconsistency, calling for an inconsistency checker or a better configuration design that does not require users to worry about such error-prone consistency constraints.

As some previous work on detecting or diagnosing misconfiguration focuses on only *single* configuration parameter mistakes, we looked into what percentages of parameter mistakes involve only a single parameter.

Table 3.5 shows the number [1] of parameters involved in the mistakes, as well as the number of parameters which were changed to fix the misconfiguration. The results indicate that there are about 23.4%∼61.2% of the parameter mistakes involve multiple parameters. Some of these cases are due to parameter value inconsistency.

In comparison, about 14.9%∼34.7% of the examined misconfigurations require fixing multiple parameters. For example, sometimes the performance of a system could be influenced by several parameters. To gain a satisfying amount of performance, all these parameters need to be considered and set correctly.

The results above can give researchers and engineers some rough ideas about what percentage of misconfigurations some particular technique (proposed either previously or in the future) are applicable.

[1] Please note these numbers may not be the same because a mistake may involve two parameters, but can be fixed by changing only one parameter.

System	Number of Involved Parameters		
	One	Multiple	Unknown
COMP-A	117(47.6%±6.1%)	117(47.6%±6.1%)	12(4.8%±2.6%)
CentOS	30(71.4%±12.8%)	10(23.8%±12.1%)	2(4.8%±6.0%)
MySQL	35(74.5%±11.0%)	11(23.4%±10.7%)	1(2.1%±3.6%)
Apache	31(62.0%±13.0%)	16(32.0%±12.4%)	3(6.0%±6.3%)
OpenLDAP	18(36.7%±11.1%)	30(61.2%±11.2%)	1(2.0%±3.2%)
System	**Number of Fixed Parameters**		
	One	**Multiple**	**Unknown**
COMP-A	189(76.8%±5.1%)	44(17.9%±4.7%)	13(5.3%±2.7%)
CentOS	33(78.6%±11.7%)	7(16.7%±10.6%)	2(4.8%±6.1%)
MySQL	39(83.0%±9.5%)	7(14.9%±9.0%)	1(2.1%±3.6%)
Apache	33(66.0%±12.7%)	14(28.0%±12.0%)	3(6.0%±6.3%)
OpenLDAP	29(59.2%±11.3%)	17(34.7%±11.0%)	3(6.1%±5.5%)

Table 3.5: **The number of parameters in the configuration parameter mistakes.**

Finding: The majority (36.7%∼74.5%) of parameter mistakes can be diagnosed by considering only one parameter, and an even higher percentage(59.2%∼83.0%) of them can be fixed by changing the value of only one parameter.

Finding: However, a significant portion (23.4%∼61.2%) of parameter mistakes involve more than one parameter, and 14.9%∼34.7% require fixing more than one parameter.

The problem domains that each parameter mistake belongs to is also an interesting question since it can warn the developers when they are designing or implementing the configuration logic of certain functional module in the system. We decide the domain based on the functionality of the involved parameter. Four major problem domains are observed: *"network"*, *"permission/privilege"*, *"performance"*, and *"devices"*. Overall, 18.3% of examined parameter mistakes relate to how the network is configured; 16.8% relate to permission/privilege; 7.1% relate to performance adjustment. For the COMP-A systems and CentOs (the OSes), 8.5%∼26.2% of examined parameter mistakes are about device configurations. There are no device configuration issues in other systems that we studied, since these systems are running on a higher level.

3.5.3 Value Inconsistency

As discussed in Chapter 3.5.2, a sizable amount of parameter mistakes are due to the fact that certain constraints/rules among configuration parameters are violated. We call this type of parameter mistakes "value inconsistency". According to Table 3.4, 4.7%∼29.6% of parameter mistakes are due to this type of problem. We focus on this type of misconfiguration is this section. Besides, we also discuss the reasons that lead to these errors as well as the potential solutions.

Users edit configuration parameters in order to fulfill a particular goal. For example, the goal could be to turn on a certain feature (e.g. virus check), to let the system run in a particular mode (e.g., multithreaded mode) or something else. Sometime this is of low complexity, since there may be only one configuration parameter which relates to the goal that users want to achieve (as shown in Table 3.5). Therefore, users only need to be aware of that parameter to get their business done. However, there are a substantial large number of cases where more than one parameters need to be considered in order to meet users' goal (also as shown in Table 3.5), which makes the tasks of configuration complex and error prone.

Figure 3.3 gives the examples of two configuration scenarios that we discussed above. In Figure 3.3(a), the parameter $cifs.unix\_security$ is a parameter designed for handling a special case in COMP-A's storage systems. The permission of files would be lost if the files that originally have some UNIX permissions are opened and saved in Windows environment. If this parameter is set with "on", the lose of permission information can be avoided. To preserve the UNIX permission information, users only need to be aware of this parameter and no other parameters will affect this parameter. Figure 3.3(b) demonstrates a case how the name-based virtual host needs to be configured in Apache web server. In order to configure a name-based virtual host, minimally two parameters (or directives) *NameVirtualHost* and *VirtualHost* need to be configured. NameVirtualHost specifies the ip (plus optional port) on which the name-based virtual hosts will be on. VirtualHost specifies the detail of each virtual host. Both NameVirtualHost and VirtualHost need to be configured, otherwise virtual host will not work. Moreover, the ip address specified in NameVirtualHost and VirtualHost should be identical. If they mismatch, virtual host will not work as well.

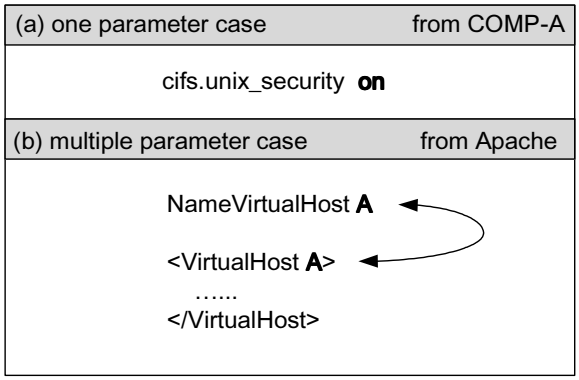

Figure 3.3: **One parameter case v.s. multiple parameter case.**

In the real world, there are many occasions similar to the example shown in Figure 3.3(b). This reveals a reality that for many configuration parameters, there are some relation constraints for the value of these parameters. The relation constraints can be defined as follows:

Relation constraint the value of parameter B is a function f of the value of parameter A.

When the function f is an identity function, it means the parameter A and parameter B should point to the same value. The function f is not necessarily just identity function. It can be any function. It could be that the value of parameter B should be a certain times of that of parameter A or some other more complex functions. Therefore, if these relation constraints are not honored, value inconsistency mistakes will happen.

It is obvious for any value inconsistency case, there must be at least two configuration parameters involved, which increase the chances for users to make mistakes especially when they are not aware of these relation constraints. This can be viewed as *"configuration vulnerabilities"*. They are similar to the general software vulnerability that incurs remote attacks and exploits.

Figure 3.4 shows six value inconsistency cases. Based on the scope of the configuration files, we can further classify these inconsistency errors into two categories. One category is *"Local"* where the relation constraint is defined on the parameters within one configuration file for an application. Figure 3.4(a), (b), and (c) belong to this category and only one configuration file is involved in those examples. The other category is *"Remote"* where the relation constraint is defined on the parameters located in the configuration files from different applications, or even on different machines. Figure 3.4(d), (e), and (f) belong to this category.

We can also look at these cases from another perspective. If we classify them based on the characteristics of the constraints, we can have the following categories:

Textually same The values of the parameters should be textually same (e.g., same ip address) or they share some textually common part.

Cooperative To enable some feature, the values of parameters need to stay in a cooperative way.

Numerical There is some numerical relation between the values of parameters. The relation could be "*less than*", "*within certain range*", "*A should be four times of B*", etc.

Others The constraint which is different from the above three.

Therefore, Figure 3.4(a), (e), and (f) belongs to *textually same*. Figure 3.4(b) and (c) fit in the categories of *cooperative*. Figure 3.4(d) is one example of *numerical*, more specifically, "*less than*".

The key to detect these inconsistency errors are those relation constraints. However, developers may not implement the checks against these relation constraints when they are writing code. We guess there could be multiple reasons.

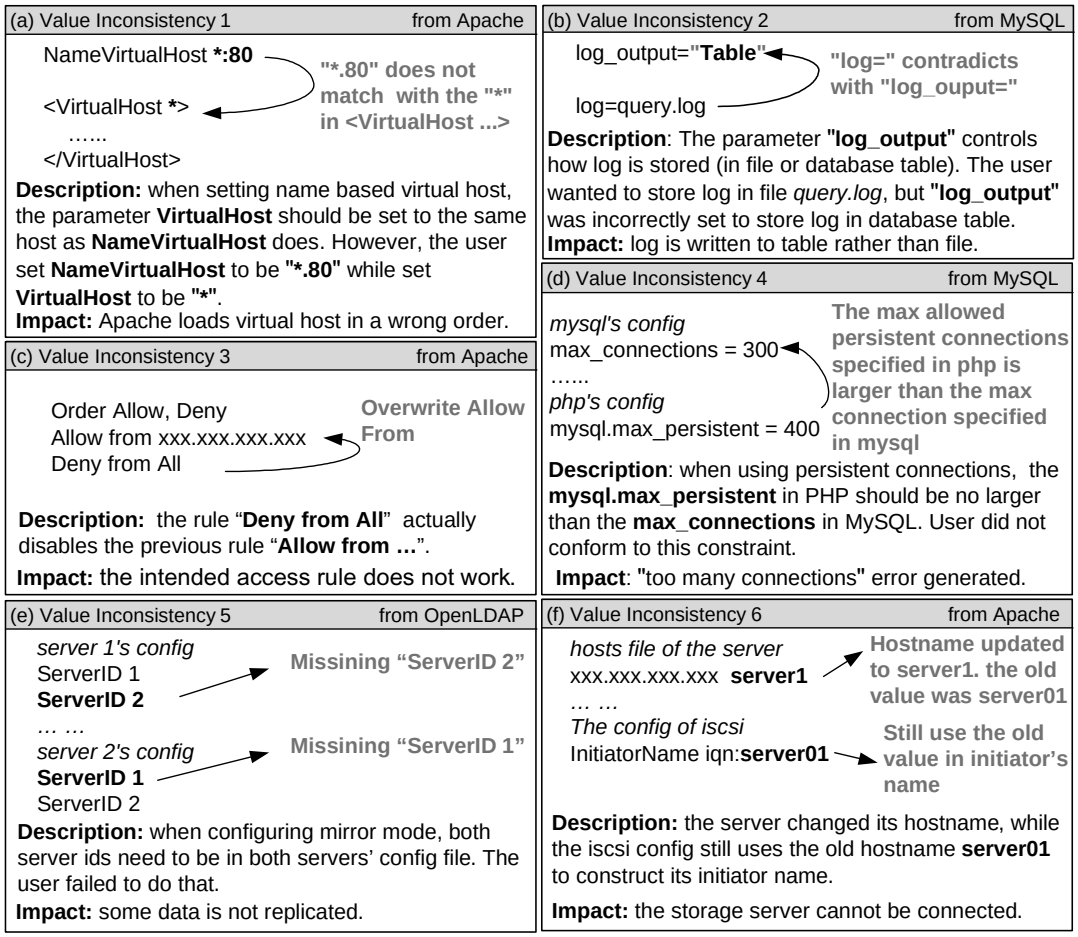

Figure 3.4: **Some value-based inconsistency mistake examples.**

Developers could forget to add the checks, could choose to not add the check due to some design considerations or even developers themselves are not aware of such relation constraints. It could also be due to the mismatch of design specification and the actual implementation. On the other side, though some of these constraints are documented in the software manual, the developers can not require users to read every word of the manual. It is a truth that people still make mistakes even though some constraints are indeed lying in the manual.

Sometimes some constraints are not documented and become "hidden" constraints which might only be aware of by some administrator gurus. Even worse, the manual could be wrong about some constraints. In Apache's configuration manual, the entry *Options* is said to be possible to be configured under the context of the virtual host directive. However, in the source code *Options* will be ignored if it is configured under the context of the virtual host directive. This is apparently against the manual.

This suggests we should not merely rely on human and manuals. Instead, tools that automatically extract

and check these constraints will be highly preferred. To build such tools, it is good to know the distribution of all these constraint violations. Table 3.6 just provides such information.

System	Textually Same	Cooperative	Numerical	Other
COMP-A	44	15	9	5
CentOS	2	3	0	1
MySQL	4	1	1	1
Apache	3	2	1	1
OpenLDAP	3	3	0	0

Table 3.6: **The relation constraints violated in the applications we examined.**

The majority (56.6%) of value inconsistency mistakes that we examined violate the "textually same" constraint. For this type of mistakes, it might be possible to extract these constraints by using a simple heuristic. The heuristic is that the parameters that have the same values textually or share some common part textually may have certain relations. Then we can group them together. This heuristic works well for the parameters whose values are ip address, host names, file paths, etc. Among all the violation to the "textually same" constraint, 58.9% (33 out of 56) of the cases are related to such type of parameters. With a large amount of configuration files obtained, data mining techniques could extract these types of constraints with a low false positive rate. For the rest of the "textually same' violations, the parameter involved usually have value type as *enum* or *integer*. The previous heuristic may not work for these cases. The small value space may introduce collision which is considered as false positive. For example, many parameters with *enum* type can have the same value as "Yes" or "No", which does not mean these parameters are related to each other.

Beside using techniques like data mining, we can also leverage program analysis to extract the constraint. The assumption here is if there is some relation between parameters, the relation should be reflected in source code in certain ways.

From MySQL mysqld.c

```
if (opt_logname && !(log_output_options & LOG_FILE)
 &&!(log_output_options & LOG_NONE))
{ ..... }
```

Figure 3.5: **A code snippet in MySQL which is related to 3.4(b).** Here the variables in bold are the corresponding variables of parameter *log_output* and *log*.

As shown in Figure 3.5, for the two parameters *log_output* and *log* that have a relation between each other, their corresponding variables in the source code are actually in the condition of the same *if* clause. This is just one type of the source code patterns. There might be many other source code patterns which suggest

the relations between parameters. However, program analysis is difficult to be applied across applications. Table 3.7 reveals that among all value-based inconsistency mistakes, 74.7% (74 out of 99) of them involve multiple applications. Therefore, for these mistakes, more sophisticated techniques may need to be studied.

Apps	Local	Remote	Total
COMP-A	11	62	73
CentOS	2	4	6
MySQL	4	3	7
Apache	5	2	7
OpenLDAP	3	3	6
Total	25	74	99

Table 3.7: **The scope of value inconsistency mistakes for all the examined applications.** *Local* means the related parameters are just within the configuration files of the same application. *Remote* means they are across applications.

3.5.4 Software Incompatibility

Besides parameter-related mistakes, software incompatibility is another major cause of misconfigurations (up to 18.3%, see Table 3.3). A complex software system involves multiple inter-dependent components with different versions. Software incompatibility issues refer to improper combinations of components or their versions. They could be caused by incompatible libraries, applications or even operating system kernels. For example, in one of the studied cases, the user failed to install the "*mod_proxy_html*" module because the existing "*libxml2*" library was not compatible with this module.

In another example, a user cannot make the PHP LDAP extension work because of one incompatible library. To install PHP LDAP extension on Windows is a multi-step process and missing any one of them will cause a failure. The mistake this user made was that the one of dynamic-link libraries, `php_ldap.dll`, was not the thread-safe version while other existing libraries were.

One way to deal with software compatibility issues is better software packaging. For example, modern software package management systems, such as RPM [12] and Debian dpkg [3], automatically detect incompatible or missing dependencies and update them. This may work well for systems with a standard package. For systems that require multiple applications from different vendors to work together, it is more challenging.

An alternative to package management systems is self-contained packaging, i.e. integrating dependent components into one installation package and minimizing the requirements on the target system. To further reduce dependencies, one could deliver a system as virtual machine images (e.g., Amazon Machine Image running in Amazon EC2) or appliances (e.g., COMP-A's storage systems or Google Search Appliance). The latter may even eliminate hardware compatibility issues.

There have been attempts on improving software compatibility by testing software in different systems [105, 22], but it seems few proposed techniques have been used in real world. For complex systems, support matrices [5, 4] are used to guide configurations. Support matrices are documents that enumerate acceptable combinations of compatible components.

One may think that system upgrades are more likely to cause software incompatibility issues, but our study found that only 11.3% of the software incompatibility issues are caused by system upgrades. One of the possible reasons is that people already put great effort to assure software upgrades proceed properly. For example, COMP-A offers a tool called Upgrade Advisor [97] to help upgrade. It creates an easy-to-understand report of all known compatibility issues, and recommends ways to resolve them.

3.5.5 Component Misconfiguration

Component misconfigurations are configuration errors that are neither parameter mistakes nor compatibility issues. 8.3%∼14.5% of our examined misconfigurations are of this category. Here we further classify them into the following five subtypes based on root causes:

Missing component Certain components (modules or libraries) are missing.

Placement Certain files or components are not in the place expected by the system.

File format The format of certain file is not acceptable to the system (For example, an Apache web server on a Linux host cannot load a configuration file because it is in the MS-DOS format with unrecognized new line characters).

Insufficient resource The available resources are not enough to support the system functionality (e.g., not enough disk space).

Stale data Stale data in the system do not support the new configuration.

Subtype	Number of Cases
Missing component	15(25.9%)
Placement	13(22.4%)
File format	3(5.2%)
Insufficient resource	15(25.7%)
Stale data	3(5.2%)
Others	9(15.5%)

Table 3.8: **Subtypes of component misconfigurations.**

Table 3.8 shows the distribution of the subtypes of component misconfigurations. "Missing component", "placement" and "insufficient resource" are the three major causes.

A missing component can prevent the functionality that depends on this component from working or even prevent the whole system from starting. Sometimes due to resource limitation, not all modules are loaded or installed at the first place. For example, some modules are loaded on demand to save memory. But regardless whether the user will use certain features or not, it is always recommended to check if all the required modules are present and issue warnings if some of them are missing. A better alternative is that the system will automatically load the component and then continue.

File placement is also needed to be watched. It creates issues for some applications where the position of certain kind of files are critical. For example, if the data files and log files of a database are not placed properly, performance can be greatly affected. For this kind of problems, the checking of such constraints should be forced at the first place.

Besides, it is good to monitor the utilization of disk space and give warning when space is close to full. But a more fundamental way is to have stressing testing specially on the environment with little free disk space before the shipment of the software. This is especially important and beneficial for storage systems.

The three stale data cases all caused severe problems: The two MySQL cases are that user could not log in after reinstallation because the authentication of previous installation was not deleted; the one COMP-A case was that a disk was added to a new filer with the undeleted ownership information. Those information caused the storage system not work correctly.

3.5.6 Mistake Location

When diagnosing misconfiguration problems, one tends to focus on the target application (i.e the application where the first symptom is observed). However, we find that misconfiguration problems can actually be located outside of the target application. We categorize the misconfiguration cases where the error is located outside of the target application as follows:

OS-FS Cases where the error is located in file system, including file system capacity, file/directory permission, file/directory location, etc.

OS-Module Cases where the error is located in operating system modules, including kernel modules, security modules (such as SELinux), etc.

Network Cases where the error is located in network configuration, including IP table, IP/hostname configuration, firewall configuration, etc.

Other application Cases where the error is located in other applications or libraries.

Environment other environment like DNS service.

System	Inside	FS	OS-Module	Network	Other App	Environment	Others
COMP-A	132(42.7±3.0%)	23(7.4±1.6%)	3(1.0±0.6%)	53(17.2±2.3%)	82(26.5±2.7%)	5(1.6±0.8%)	11(3.6±1.1%)
CentOS	26(43.3±4.0%)	2(3.3±1.4%)	12(20.0±3.2%)	4(6.7±2.0%)	11(18.3±3.1%)	2(3.3±1.4%)	3(5.0±1.8%)
MySQL	27(49.1±3.2%)	10(18.2±2.5%)	6(10.9±2.0%)	1(1.8±0.9%)	6(10.9±2.0%)	4(7.3±1.7%)	1(1.8±0.9%)
Apache	47(78.3±3.1%)	3(5.0±1.7%)	3(5.0±1.7%)	3(5.0±1.7%)	3(5.0±1.7%)	0	1(1.7±1.0%)
OpenLDAP	39(62.9±3.4%)	2(3.2±1.3%)	1(1.6±0.9%)	0	17(27.4±3.3%)	1(1.6±0.9%)	2(3.2±1.3%)

Table 3.9: **The location of errors.**

Table 3.9 shows the distribution of configuration error locations. Naturally, most misconfigurations are contained in the target application itself. However, many misconfigurations also span to places beyond the application. The administrators also need to consider other parts of the system, including file system permissions/capacities, operating system modules, other applications running in the system, network configuration, etc. So looking at only the application itself is not enough to diagnose and fix many configuration errors.

Finding: Although most misconfigurations are located within each examined application, a significant portion (21.7%∼57.3%) involve configurations beyond the application itself or span over multiple hosts.

3.5.7 Hardware Misconfiguration

As mentioned in Chapter 3.1, configuration errors also contains hardware misconfigurations. In previous sections, we had studied the misconfigurations manifested at software layer. In this section, we briefly discuss the misconfigurations occurred at hardware layer.

For simple systems, issues that relate to hardware configurations are seldom reported, since these systems are usually running on a higher layer so that they rarely need to worry about the interaction with hardware. Therefore, we just focused on the hardware misconfiguration from COMP-A.

From the 1000 cases that we sampled from COMP-A's customer issue database, we found 26 hardware misconfiguration cases. The amount is at the similar level with "Software Incompatibility" and "Component Misconfiguration" (32 and 31, respectively). These cases can be further classified into three categories:

Not supported device Certain device is not supported in current configuration (e.g., a special network card is not supported in certain type of motherboard).

Wrong slots/cabling A device is plugged into a wrong slot on the mainboard or the interconnection (e.g., network cable) is incorrectly connected.

Wrong configuration There are no unsupported device or incorrect cabling, but certain hardware configuration is not set in a right way. These are the configurations that can only be adjusted at hardware layer (e.g., a button is not on the right position).

Subtype	Number of Cases
Unsupported Device	12(46.1%)
Wrong Slots/Cabling	10(38.5%)
Wrong Configuration	4(15.4%)

Table 3.10: **Subtypes of hardware misconfigurations in COMP-A.**

As shown in Table 3.10, close to half (46.1%) of the hardware miconfigurations from COMP-A are problems with unsupported devices. Many of such cases are due to the use of some unsupported hard disks. We also observed cases such as unsupported network switch and system cards. Though they account for the big part of hardware misconfigurations, for most of them (75.0%), explicit error messages that pinpoint the root cause of the failures will be generated when these misconfigurations are observed. Therefore it is easy for support engineers to handle such failures.

38.5% of the hardware miconfigurations from COMP-A are due to putting devices in wrong slots or incorrect cabling. Comparing to "unsupported devices", these are more difficult to diagnose. Only for 20% of them, we will see explicit error messages about the root cause. Among these cases, wrong cabling is a more severe problem. With the prevalence of data centers, the interconnection in data centers will be extremely complex with lots of network cables. All of them are needed to be connected by human. Human errors are highly likely considering such scale. When wrong cabling happens, it also takes a lot of human effort to resolve. Some support engineers from COMP-A complained to us that the wrong cabling is usually the hardest case that they had encountered in the field. To solve this problem, it is good to design some detection logic into the hardware so that the connection of cabling can be easily inspected. Therefore, miscabling could be alerted automatically.

Besides, a small portion (15.4%) of cases are due to that some buttons or switches are not in right position. Developers can try to expose the status of these hardware "knobs" to the upper level software layer. Then we can build some detector to catch these mistakes.

3.6 System Reaction to Misconfiguration

In this section, we examine system reactions to misconfigurations, focusing on whether the system detects the misconfiguration and on the error messages issued by the system.

3.6.1 Do Systems Detect and Report Configuration Errors?

Proactive detection and informative reporting can help diagnose misconfigurations more easily. Therefore, we wish to understand whether systems detect and report configuration errors. We divide the examined cases into three categories based on how well the system handles configuration errors (Table 3.11). Cases where the systems and associated tools detect, report, recover from (or helps the user correct) misconfigurations may not be reported by users. Therefore, the results in this section may be especially skewed by our available data. Nevertheless, there are interesting findings that arise from this analysis.

System	Pinpoint Reaction	Indeterminate Reaction	Quiet Failure	Unknown
COMP-A	48(15.5±2.2%)	153(49.5±3.0%)	74(23.9±2.6%)	34(11.0±1.9%)
CentOS	7(11.7±2.4%)	33(55.0±3.7%)	16(26.7±3.3%)	4(6.7±1.9%)
MySQL	4(7.2±1.7%)	26(47.3±3.2%)	13(23.6±2.8%)	12(21.8±2.7%)
Apache	8(13.3±2.6%)	28(46.7±3.8%)	16(26.7±3.4%)	8(13.3±2.6%)
OpenLDAP	9(14.5±2.6%)	28(45.2%±3.7%)	14(22.6±3.1%)	11(17.7±2.8%)

Table 3.11: **The number of cases in each category of system reaction.**

System	Mysterious Symptoms w/o Message
COMP-A	26(8.4±1.7%)
CentOS	4(6.7±1.9%)
MySQL	9(16.4±2.4%)
Apache	3(5.0±1.7%)
OpenLDAP	3(4.8±1.5%)

Table 3.12: **The number of cases that cause mysterious crashes, hangs, etc. but do not provide any messages.**

We classify system reactions into *pinpoint reaction*, *indeterminate reaction*, and *quiet failure*.

A pinpoint reaction is one of the best system reactions to misconfigurations. The system not only detects a configuration error but also pinpoints the exact root cause in the error message (see a COMP-A example in Figure 3.6). As shown in Table 3.11, more than 85% of the cases do *not* belong to this category, indicating systems may *not* react in a user-friendly way to misconfigurations. As previously discussed, the study includes only reported cases. Therefore some misconfigurations with good error messages may have already been solved by users themselves and thus not reported. So in reality, the percentage of pinpoint reaction to misconfiguration may be higher. However, considering the total number of misconfigurations in the sources we selected is very large, there are still a significant number of misconfigurations for which the examined systems do not pinpoint the misconfigurations.

An indeterminate reaction is a reaction that a system does provide some information about the failure symptoms (i.e., manifestation of the misconfiguration), but does not pinpoint the root cause or guide the

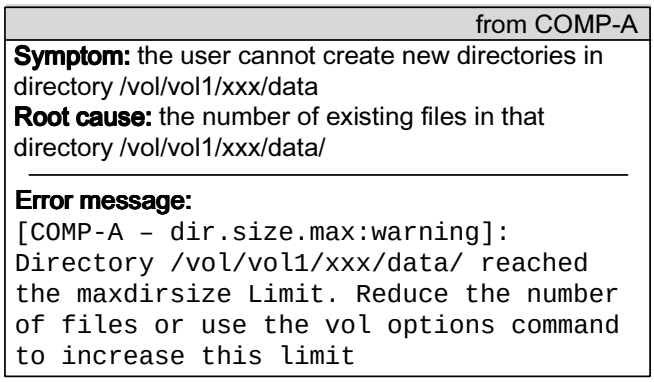

Figure 3.6: **A misconfiguration case where the error message pinpoints the root cause and tells the user how to fix it.**

user on how to fix the problem. 45.2%∼55.0% of our studied cases belong to this category.

A quiet failure refers to cases where the system does not function properly, and it further does not provide any information regarding the failure or the root cause. 22.6%∼26.7% of the cases belong to this category. Diagnosing them is very difficult.

Finding: 7.2%∼15.5% of the studied misconfiguration problems provide explicit messages that pinpoint the configuration error.

Things can be even worse when the misconfiguration causes the system to misbehave in a mysterious way (crash, hang, etc.) just like software bugs. We found that in most systems, this occurs in 5%∼8% of the cases we studied (Table 3.12).

Why would misconfigurations cause a system to crash or hang unexpectedly? The reason is intuitive: since configuration parameters can also be considered as a form of input, if a system does not perform validity checking and prepare for illegal configurations, it may lead to system misbehavior. We describe two such scenarios below.

Crash example: a web application used both *mod_python* and *mod_wsgi* modules in Apache httpd server. These two modules used two different versions of Python, which caused segmentation fault errors when trying to access the web page.

Hang example: a server was configured to authenticate via LDAP with the *hard* bind policy, which made it keep connecting to the LDAP server until it succeeded. However, the LDAP server was not working, so the server hung when the user added new accounts.

Such misbehavior is very challenging to diagnose because users and support engineers may suspect these unexpected failures to have been caused by a bug in the system instead of a configuration issue (of course,

one may argue that, in a way it can also be considered to be a bug). If the system is built to perform more thorough configuration validity-checking and avoid misconfiguration-caused misbehavior, both the cost of support and the diagnosis time can be reduced.

Finding: Some misconfigurations have caused the systems to crash, hang or have severe performance degradation, making failure diagnosis a challenging task.

We further studied if there is certain correlation between the type of misconfiguration and the difficulty for systems to react. We calculated the ratio of cases that have *pinpoint reaction* to the total cases in each type of misconfigurations. The result shows it is more difficult to have appropriate reaction for software incompatibility issues. Only 9.3% of all the incompatibility issues have "pinpoint reaction", while the same ratio for parameter mistakes and component misconfigurations is 14.3% and 15.5% respectively. This result is reasonable since global knowledge (e.g., the configuration of different applications) is often required to decide if there are incompatibility issues.

3.6.2 System Reaction to Illegal Parameters

Cases with illegal configuration parameters (defined inChapter 3.5.2) are usually easier to be checked and pinpointed automatically.

System	Pinpoint Reaction	Indeterminate Reaction	Quiet Failure	Unknown
COMP-A	25(18.9%)	57(43.2%)	27(20.5%)	23(17.4%)
CentOS	4(25.0%)	7(43.8%)	5(31.3%)	0
MySQL	1(4.3%)	13(56.5%)	3(13.0%)	6(26.1%)
Apache	5(21.7%)	9(39.1%)	4(17.4%)	5(21.7%)
OpenLDAP	7(26.9%)	11(42.3%)	4(15.4%)	4(15.4%)

Table 3.13: **How do systems react to illegal parameters?** The reaction category is the same as in Table 3.11.

However, we found that systems do not detect and pinpoint a majority of these configuration mistakes, as shown in Table 3.13.

Finding: Among 220 cases with illegal parameters that could be easily detected and fixed, only 4.3%∼26.9% of them provide explicit messages. Up to 31.3% of them do not provide any message at all, unnecessarily complicating the diagnosis process.

3.6.3 Impact of Messages on Diagnosis Time

Do good error messages help engineers diagnose misconfiguration problems more efficiently? To answer this question, we calculated the diagnosis time, in the unit of hours, from the time when a misconfiguration problem was posted to the time when the correct answer was provided. Table 3.14 summarizes the data.

System	Explicit Message	Ambiguous Message	No Message
COMP-A	1x	13x	14.5x
CentOS	1x	3x	5.5x
MySQL	1x	3.4x	1.2x
Apache	1x	10x	3x
OpenLDAP	1x	5.3x	2.5x

Table 3.14: **The *median* of diagnosis time for cases with and without messages (time is normalized for confidentiality reasons).** "Explicit message" means that the error message directly pinpoints the location of the misconfiguration. The median diagnosis time of the cases with explicit messages is used as base. "Implicit message" means there are messages but they do not directly identify the misconfiguration. "No message" is for cases where no messages are provided.

Table 3.14 presents that the misconfiguration cases with explicit messages are diagnosed much faster, with the median of diagnosis time of less than 4 hours. Otherwise, engineers have to spend much more time on diagnosis, where the median of the diagnosis time is up to 14.5 times longer.

Finding: Messages that pinpoint configuration errors can shorten the diagnosis time 3 to 13 times as compared to the cases with ambiguous messages or 1.2 to 14.5 times as compared to the cases with no messages.

To improve error reporting, two types of approaches can be adopted. A white-box approach [108], uses program analysis to identify the state that should be captured at each logging statement in source code to minimize ambiguity in error messages. When source code is not available, a black-box approach, such as Clarify [45], can be taken instead. Clarify associates the program's runtime profile with ambiguous error report, which enables improved error reporting.

Interestingly, for some of the systems (Apache, MySQL and OpenLDAP), engineers seem to spend more time (2∼4 times longer) diagnosing cases with ambiguous messages than cases with no messages at all. There are several potential reasons. First, incorrect or irrelevant messages can sometimes mislead engineers, directing them down a wrong path. Figure 3.7 shows such an example. Based on the message provided by the client, both the support engineers and the customers thought the problem was on the client end, so they made several attempts to set certificates, but the root cause turned out to be a problem in the configuration on the server side. This indicates that the accuracy of messages is critical to the diagnosis process. Providing

misleading messages may be worse than providing nothing at all.

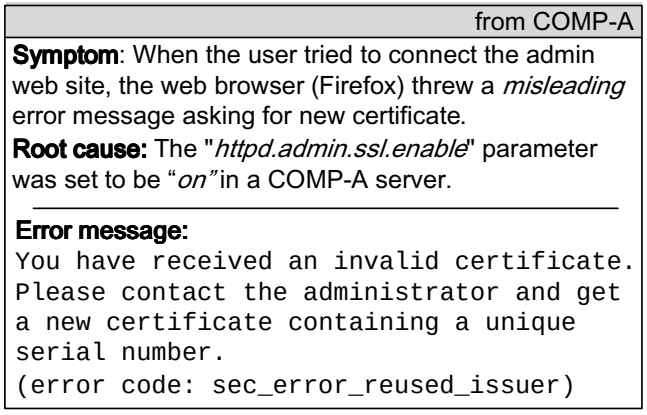

Figure 3.7: **A misconfiguration case where the error message misled the customer and the support engineers.**

Second, in some cases, symptoms and configuration file content are already sufficient for support engineers or experts to resolve the problem. For these cases, whether there are error messages is less important. For example, many cases from MySQL related to performance degradation do not have error messages, but it was relatively easy for experts to solve those problems by looking at only configuration file. However, even for these cases, if the system could give good quality messages, users could have solved the problems by themselves.

Finding: Giving an irrelevant message may be worse than not giving message at all for diagnosing misconfiguration. Some irrelevant messages could mislead users to chase down the wrong path. In three of the five studied systems, statistic data shows that ambiguous messages may lead to longer diagnosis time compared to not having any message.

We further did a preliminary study on what kind of error messages are more useful in reducing diagnosis time. Specifically, we read through the misconfiguration cases that have explicit messages and are parameter mistakes (a total of 62 cases). Besides that all these cases pinpoint the root cause of the failure (which is our definition of "*explicit*"), 69.4% of them further mention the parameter name in the message; 6.5% of even further point out the parameter's location within the configuration file. However, we did not find strong correlation between the diagnosis time and these extra information (e.g., parameter name) in the explicit messages. A more comprehensive study on this topic will remain as our future work.

3.7 Causes of Misconfigurations

3.7.1 When Did Misconfigurations Happen?

There are many ways to look at the reasons to misconfiguration. Here, we examined only a couple of them. Misconfiguration problems could happen when the user uses (or configures) a feature for the first time, or when a previous working feature does not work any more. Based on this, we categorized the misconfiguration cases into two categories (Table 3.15):

Used-to-work The system used to work but does not work any more due to various changes.

First-time use The misconfiguration happens at the user's first attempt to access certain functionality.

System	Used-to-Work	First-Time Use	Unknown
COMP-A	100(32.4±2.8%)	165(53.4±3.0%)	44(14.2±2.1%)
CentOS	10(16.7±3.0%)	40(66.6±3.8%)	10(16.7±3.0%)
MySQL	3(5.5±1.5%)	45(81.8±2.5%)	7(12.7±2.2%)
Apache	2(3.3±1.4%)	40(66.7±3.6%)	18(30.0±3.5%)
OpenLDAP	2(3.2±1.3%)	57(91.9±1.6%)	3(4.8±1.6%)

Table 3.15: **The number of misconfigurations categorized by "used-to-work" and "first-time use".**

There are cases that we do not have adequate information to decide which category it belongs. We marked those cases as "Unknown".

One may think that most misconfigurations happen when users configure a system for the first time. As our results show, it is indeed the case for relatively simple systems (MySQL, Apache and OpenLDAP). The cause for the misconfigurations during "*First-time use*" can be the inadequate knowledge of personnel. Therefore users made some mistakes and the system become misconfigured. The potential solution for this types of cases could be have better training of the personnel, building better knowledge base so users can easily look up, or designing checkers which can detect some configuration errors before running the system. A more fundamental way is to have novel design of configurations that is of low complexity and really easy-to-use. Sometimes, there is no problem with user but with the user manuals. There are indeed some manuals which contains inconsistent content [86]. We also found some manuals have certain wrong entries (as discussed in Chapter 3.5.3). User may make mistake when following these "wrong" manuals.

However, for more complex systems, COMP-A and CentOS, a significant portion (16.7%∼32.4%) of the misconfigurations actually happen in the middle of the system's lifetime. There could be two major reasons. First, these systems have much more frequent changes (upgrades, reconfiguration, etc.) in their lifetime. Second, the configuration is more complicated so it takes a long time for users to master.

Finding: For simpler systems, the majority of misconfigurations are related to first-time use of certain functionality. For more complex and larger systems, a significant percentage (16.7%∼32.4%) of the misconfigurations were introduced into systems that used to work fine.

3.7.2 Why Does My System Stop Working?

To further zoom into the causes for the "used-to-work" cases, we categorized the 100 cases of this category from COMP-A based on their root causes (Figure 3.8).

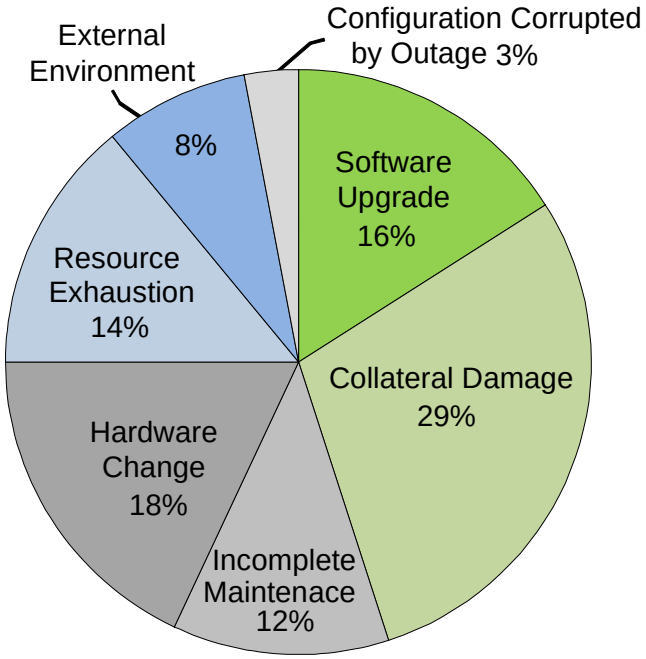

Figure 3.8: **The cause distribution for the "used-to-work" misconfigurations at COMP-A.**

"Collateral damage" refers to cases when users made configuration changes for some new functionality but accidentally broke existing functionality. It accounts for 29.0% of the "used-to-work" cases from COMP-A. To avoid such collateral damages, it might be useful if users can be warned by the configuration management/change tool about the side-effects of their changes.

"Incomplete maintenance" refers to cases when some regular maintenance tasks introduced configuration changes breaking existing functionality. 12.0% of the "used-to-work" cases from COMP-A belong to this category. For example, when an administrator does a routine periodic password change to certain accounts but forgets to propagate it to all affected systems, some systems would not be able to authenticate these accounts.

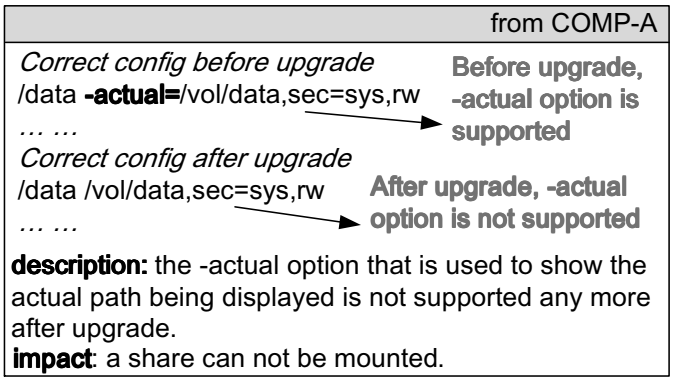

Figure 3.9: **A misconfiguration example where the syntax of configuration files has changed after upgrade.** A previously working NFS mounting configuration is no longer valid, because the option *actual* became deprecated after upgrade.

In addition, configuration could also be "corrupted by outage" (3.0%) or be modified accidentally by some (2.0%) of "software upgrades" (Figure 3.8). To sum up, *46% of the "used-to-work" misconfiguration cases from COMP-A are caused by configuration parameter changes due to various reasons including configuring other features, routine maintenance, system outages, or software upgrades.* To diagnose and fix these cases, it is useful for systems to automatically keep track of configuration changes [97], and even better, help users to pinpoint which change is the culprit like in [103].

Another major cause is "hardware change" (18.0%). When customers upgrade, replace or reorganize hardware (e.g., moving a disk from one server to another), it can cause problems if they forget to change related configuration parameters accordingly.

"Resource exhaustion" (14.0%) can also affect a previously working system. For example, in one of the studied cases, a database system hung and did not work properly even after rebooting because the data disks became full.

Moreover, "external environment" changes could also be harmful to previously working systems. They account for 8.0% of "used-to-work" cases from COMP-A. For example, in one of the studied cases, a system suffered from severe performance degradation because its primary DNS server went offline accidentally. Such changes are error-prone and problematic, because, in a data center, typically different systems are managed by different administrators who may not communicate with each other in a timely manner about their changes.

"Software upgrade", as one may expect, is another major cause of misconfigurations that break a previously working system. It accounts for 16% of the "used-to-work" cases from COMP-A. We further subcategorize it into three types. First, a new software release may have changed the configuration file syntax

Configuration Syntax Being Changed	Configuration Being Modified	Incompatible Upgrade	Total
5%	2%	9%	16%

Table 3.16: **The breakdown of "used-to-work" misconfiguration cases from COMP-A caused by software upgrade.** Percentages are calculated based on the total number of "used-to-work" cases.

or format requirements, making the old configuration file invalid. Figure 3.9 gives such an example. Second, some automatic upgrade processes may silently modify certain configuration parameters (e.g. set them to default values) without users' awareness. Third, software upgrades may cause incompatibilities among components.

Addressing the misconfiguration errors introduced by software upgrade requires the efforts from both developers and users (administrators). On one hand, developers should test the influence of the program behavior change on previous configuration. This can be achieved by regression testing with different combination of configurations. Developers need also explicitly document the changes of the syntax of configuration language and highlight them to the users. On the other hand, it may be a good practice to keep a copy of previous working configuration before upgrade in case that the configuration might be messed up by upgrade. Users may also need to be alert to the syntax changes of configuration language. For the compatibility issues, users should make sure all the modules that need to be upgraded are actually upgraded. Systems should provide automatic upgrade tools or at least detailed upgrade instructions [97, 34]. The upgrade process should also take users' existing configurations into consideration.

Finding: By looking into the 100 "used-to-work" cases (32.4% of the total) at COMP-A, 46% of them are attributed to configuration parameter changes due to routine maintenance, configuring for new functionality, system outages, etc, and can benefit from tracking configuration changes. The remainder are caused by non-parameter related issues such as hardware changes (18%), external environmental changes (8%), resource exhaustion (14%), and software upgrades(14%).

3.8 Impact of Misconfigurations

We analyzed the severity of customer-reported issues from COMP-A (Chapter 3.4) and found that a large percentage (31%) of high-impact customer issues were related to system configuration. In this section, we analyze the severity of the specific misconfiguration cases used in our study, particularly from the viewpoint of system availability and performance. We divide the misconfiguration cases into three categories, as shown in Table 3.17: (1) the system becomes "*fully unavailable*"; (2) the system becomes "*partially unavailable*", i.e.

it cannot deliver certain desired features; and (3) the system suffers from severe "*performance degradation*". We do expect the results to be skewed towards the more severe cases which caused users to report them more often than simpler cases.

System	Fully Unavailable	Partially Unavailable	Performance Degradation
COMP-A	41 (13.3±2.1%)	247 (79.9±2.4%)	21 (6.8±1.5%)
CentOS	12 (20.0±3.2%)	47 (78.3±3.3%)	1 (1.7±1.0%)
MySQL	15 (27.3±2.9%)	29 (52.7±3.2%)	11 (20.0±2.6%)
Apache	15 (25.0±3.3%)	44 (73.3±3.4%)	1 (1.7±1.0%)
OpenLDAP	6 (9.7±2.2%)	52 (83.9±2.7%)	4 (6.4±1.8%)

Table 3.17: **The impact distribution of the misconfiguration cases from all the studied systems.**

As the result shows, 9.7%∼27.3% of the misconfigurations cause the system to become fully unavailable. This demonstrates again that misconfiguration can be a very severe threat to system availability.

Moreover, up to 20.0% of the misconfigurations cause severe performance degradation, especially for systems such as database servers that are performance sensitive and require some non-trivial tuning based on users' particular workloads, infrastructure, and data sizes. For example, the official performance tuning guides for MySQL and Oracle have more than 400 pages, and mention tens, even hundreds of configuration parameters that are related to performance. The ratio here might be an underestimation of performance problems in the field, since some of trivial performance issues introduced by misconfiguration may not be reported by the user.

Finding: Although most studied misconfiguration cases only lead to partial unavailability of the system, 16.1%∼47.3% of them make the systems to be fully unavailable or cause visible performance degradation.

The next question is whether different types of misconfigurations have different impact characteristics. So we also examined the impact of each type of misconfiguration and the results are shown in Table 3.18.

Misconfiguration Type	Fully Unavailable	Partially Unavailable	Performance Degradation
Parameters	59 (**13.6%**)	342 (78.8%)	33 (7.6%)
Compatibility	14 (**25.9%**)	38 (70.4%)	2 (3.7%)
Component	16 (**27.6%**)	39 (67.2%)	3 (5.2%)

Table 3.18: **The impact on different types of misconfiguration cases.** The data is aggregated for all the examined systems. The percentage shows the ratio of a specific type of misconfiguration (e.g., parameter mistake) that lead to a specific impact level (e.g., full unavailability).

We found that, compared to configuration parameter mistakes, software compatibility and component configuration errors are more likely to cause full unavailability of the system. 25.9% of the software compat-

ibility issues and 27.6% of the component configuration errors make systems fully unavailable, whereas this ratio is only 13.6% for parameter-related misconfigurations.

The above results are not surprising because what components are used and whether they are compatible can easily prevent systems from even being unable to start. In contrast, configuration parameter mistakes, especially if the parameter is only for certain functionality, tend to have a much more localized impact.

In addition to having a more severe impact, compatibility and component configuration mistakes can be more difficult to fix. They usually require greater expertise from users. For example, in one of the misconfiguration cases of CentOS, the user could not mount a newly created ReiserFS file system, because the kernel support for this ReiserFS file system was missing. The user needed to install a set of libraries and kernel modules and also modify configuration parameters in several places to get it to work.

Finding: Software compatibility and component configuration errors are more likely to lead to full unavailability of the system, calling for more attention to avoid compatibility and component configuration issues.

Chapter 4

Conclusion

System reliability and manageability are the two key problems that relate to system failures. Though substantial advancement has been achieved in improving system performance, the breakthrough in improving system reliability and manageability has been less observed. With the trend that systems are becoming more and more complex, we will face even more challenges when fighting against system failures.

No matter coming up with intelligent tools to detect those errors or an innovative design to avoid those faults fundamentally, it is important to first understand the characteristics of those failures. Such a characteristics can serve as a lighthouse which give navigation to both academia and industry efforts on solving the problem of system failures.

This thesis did a comprehensive study on incorrect fixes (i.e., buggy patches) and misconfigurations. One is a major cause to system reliability and the other is a big issue in system manageability. Both of them can cause severe system failure, large financial losses or even human causality. In order to make our study as comprehensive and complete as possible, we sampled a large number of bug fixes and customer (configuration related) issues to guarantee the significance in statistics. We also chose our samples from both open source systems and commercial systems. These systems are all widely deployed and used and our samples are all real system failures that happened in the field.

For incorrect fixes, we studied their prevalence, the specific bug types that are more difficult to fix right, the common mistake patterns during bug fixing and the human factors that relate to incorrect fixes. Our findings demonstrate incorrect fix is indeed a significant problem that should draw our attention. The mistake patterns we found are very useful to build checkers that can effectively detect incorrect fixes. We also found code knowledge is an important factor that influences the quality of bug fixing. The existing bug fixing/reviewing assignment is not always assigning the right person to do the fixing and reviewing. Inspired by our findings, the commercial software vendor whose OS code we evaluated is building a tool to improve the bug fixing and code reviewing process.

For misconfigurations, we studied their ratio among all the customer issues, their different types, including both parameter mistakes and non-parameter mistakes. We also studied certain misconfiguration

patterns, how systems react to a misconfiguration, the various causes of configuration errors and the impact of misconfigurations. We found configuration related cases are very prevalent. Especially for high severity cases, configuration related cases are most reported. Most misconfigurations are due to parameter mistakes but there are still sizable amount of non-parameter mistakes that should be handled. Value inconsistency mistakes is a frequently observed type among parameter mistakes. It exposes some vulnerability in system manageability and it is easy for operator make such mistakes. We also found the quality of error messages that react to misconfiguration is very crucial to the efficacy of diagnosis. We hope that our study helps extend and improve tools that inject, detect, diagnose, or fix misconfigurations. Further, we hope that the study provides system architects, developers, and testers insights into configuration-logic design and testing, and also encourages support personnel to record field configuration problems more rigorously so that vendors can learn from historical mistakes.

References

[1] An Investigation of the Therac-25 Accidents. http://courses.cs.vt.edu/cs3604/lib/Therac_25/Therac_1.html.

[2] Coverity prevent. http://www.coverity.com/html/coverity-prevent.html.

[3] Debian package management program (dpkg). http://www.debian.org/doc/FAQ/ch-pkgtools.en.html.

[4] EMC Interoperability Support Matrices. http://www.emc.com/products/interoperability/.

[5] HP-UX 11i Support Matrix. http://h20338.www2.hp.com/hpux11i/downloads/public_hp-ux_systems_support.pdf.

[6] June 2011 | TOP500 Supercomputing Sites. http://www.top500.org/lists/2011/06.

[7] McAfee Releases Buggy Patch for VirusScan. http://news.softpedia.com/news/McAfee-Releases-Buggy-Patch-for-VirusScan-Enterprise-113749.shtml.

[8] Microsoft security bulletin. http://www.microsoft.com/technet/security/current.aspx.

[9] NetApp Protection Manager. http://www.netapp.com/us/products/management-software/protection.html.

[10] Operating System Share Over Time. http://www.top500.org/overtime/list/32/os.

[11] Reckless driving on the internet. http://www.renesys.com/blog/2009/02/the-flap-heard-around-the-worl.shtml.

[12] Rpm package manager (rpm). http://rpm.org/.

[13] Solaris. http://www.oracle.com/us/solaris/index.html.

[14] Symantec cancels buggy patch. http://news.techworld.com/personal-tech/3200465/symantec-cancels-buggy-patch/.

[15] Weka 3: Data Mining Software in Java. http://www.cs.waikato.ac.nz/ml/weka/.

[16] Federal Standard 1037C. In *Telecommunications: Glossary of Telecommunication Terms*, August 1996.

[17] Microsoft baseline security analyzer. 2008. http://www.microsoft.com/technet/security/tools/MBSAHome.mspx.

[18] After buggy patch, criminals exploit Windows flaw. http://www.infoworld.com/d/security-central/after-buggy-patch-criminals-exploit-windows-flaw-848.

[19] Paul Anderson, Patrick Goldsack, and Jim Paterson. SmartFrog meets LCFG Autonomous Reconfiguration with Central Policy Control. In *LISA*, August 2003.

[20] John Anvik, Lyndon Hiew, and Gail C. Murphy. Who should fix this bug? In *ICSE'06*.

[21] Apple updates leopard again. http://voices.washingtonpost.com/fasterforward/2008/02/apple_updates_leopardagain.html.

[22] ApTest. http://www.aptest.com/compatibility.html.

[23] Jorge Aranda and Gina Venolia. The secret life of bugs: Going past the errors and omissions in software repositories. In *ICSE*, May 2009.

[24] Mona Attariyan and Jason Flinn. Using causality to diagnose configuration bugs. In *USENIX*, June 2008.

[25] Mona Attariyan and Jason Flinn. Automating configuration troubleshooting with dynamic information flow analysis. In *OSDI*, October 2010.

[26] Adrian Bachmann, Christian Bird, Foyzur Rahman, Premkumar Devanbu, and Abraham Bernstein. The missing links: Bugs and bug-fix commits. In *FSE'10*.

[27] Marla J. Baker and Stephen G. Eick. Visualizing software systems. In *ICSE'94*.

[28] S. Beattie, S. Arnold, C. Cowan, P. Wagle, and C. Wright. Timing the application of security patches for optimal uptime. In *LISA*, November 2002.

[29] Christian Bird, Nachiappan Nagappan, Premkumar Devanbu, Harald Gall, and Brendan Murphy. Does distributed development affect software quality? An empirical case study of Windows Vista. In *ICSE'09*.

[30] Aaron B. Brown and David A. Patterson. Undo for Operators: Building an Undoable E-mail Store. In *USENIX*, June 2003.

[31] Buggy McAfee update whacks Windows XP PCs. http://news.cnet.com/8301-1009_3-20003074-83.html.

[32] A. Chou, J. Yang, B. Chelf, S. Hallem, and D. Engler. An empirical study of operating systems errors. In *SOSP*, October 2001.

[33] Andy Chou, Junfeng Yang, Benjamin Chelf, Seth Hallem, and Dawson Engler. An empirical study of operating systems errors. In *SOSP'01*.

[34] Olivier Crameri, Nikola Knezević, Dejan Kostić, Ricardo Bianchini, and Willy Zwaenepoel. Staged Deployment in Mirage, an Integrated Software Upgrade Testing and Distribution System. In *SOSP'07*, October 2007.

[35] Dawson Engler and Ken Ashcraft. RacerX: effective, static detection of race conditions and deadlocks. In *SOSP'03*.

[36] Dawson Engler, David Yu Chen, Seth Hallem, Andy Chou, and Benjamin Chelf. Bugs as deviant behavior: a general approach to inferring errors in systems code. In *SOSP'01*.

[37] More details on today's outage. http://www.facebook.com/notes/facebook-engineering/more-details-on-todays-outage/431441338919.

[38] N. Feamster and H. Balakrishnan. Detecting BGP configuration faults with static analysis. In *NSDI*, May 2005.

[39] Michael Fischer, Martin Pinzger, and Harald Gall. Populating a release history database from version control and bug tracking systems. In *ICSM'03*.

[40] David Freedman, Robert Pisani, and Roger Purves. *Statistics, 3rd Edition*. W. W. Norton & Company., 1997.

[41] Thomas Fritzy, Gail C. Murphyy, and Emily Hill. Maintaining mental models: A study of developer work habits. In *FSE*, September 2007.

[42] Keith B. Gallagher and James R. Lyle. Using program slicing in software maintenance. *IEEE Transactions on Software Engineering*, 17(8):751–761, 1991.

[43] Jim Gray. Why do computers stop and what can be done about it? In *Symp. on Reliability in Distributed Software and Database Systems*, 1986.

[44] Zhongxian Gu, Earl T.Barr, David J.Hamilton, and Zhendong Su. Has the bug really been fixed? In *ICSE'10*.

[45] Jungwoo Ha, Christopher J. Rossbach, Jason V. Davis, Indrajit Roy, Hany E. Ramadan, Donald E. Porter, David L. Chen, and Emmett Witchel. Improved Error Reporting for Software that Uses Black-Box Components. In *PLDI*, 2007.

[46] Mei-Chen Hsueh, T.K. Tsai, and R.K. Iyer. Fault injection techniques and tools. *Computer*, 30(4):75 –82, apr 1997.

[47] Intel reissues buggy patch. `http://www.pcworld.com/businesscenter/article/126918/rss.html`.

[48] A. Kappor. Web-to-host: Reducing total cost of ownership. In *Technical Report 200503, The Tolly Group*, May 2000.

[49] David Kawrykow and Martin P. Robillard. Non-essential changes in version histories. In *ICSE'11*, May 2011.

[50] Lorenzo Keller, Prasang Upadhyaya, and George Candea. ConfErr: A Tool for Assessing Resilience to Human Configuration Errors. In *DSN*, June 2008.

[51] Sunghun Kim, Kai Pan, and Jr. E. James Whitehead. Memories of bug fixes. In *FSE'06*, November 2006.

[52] Sunghun Kim, E. James Whitehead, Jr., and Yi Zhang. Classifying Software Changes: Clean or Buggy? *IEEE Trans. Software Engineering*, 34(2):181–196, March 2008.

[53] Thomas Kropf. *Introduction to Formal Hardware Verification*. Springer, 1999.

[54] Nate Kushman and Dina Katabi. Enabling Configuration-Independent Automation by Non-Expert Users. In *OSDI*, October 2010.

[55] Thomas D Latoza, Gina Venolia, and Robert Deline. Does a programmerâĂŹs activity indicate knowledge of code? In *ICSE*, May 2006.

[56] Z. Li, S. Lu, S. Myagmar, and Y. Zhou. CP-Miner: A tool for finding copy-paste and related bugs in operating system code. In *OSDI*, December 2004.

[57] Z. Li, L. Tan, X. Wang, S. Lu, Y. Zhou, and C. Zhai. Have things changed now? An empirical study of bug characteristics in modern open source software. In *ASID'06*.

[58] Z. Li and Y. Zhou. PR-Miner: Automatically extracting implicit programming rules and detecting violations in large software code. In *FSE'05*.

[59] Z. Li and Y. Zhou. Pr-miner: Automatically extracting implicit programming rules and detecting violations in large software code. In *Proc. 13th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE'05)*, September 2005.

[60] S. Lu, S. Park, E. Seo, and Y. Zhou. Learning from mistakes – a comprehensive study on real world concurrency bug characteristics. In *ASPLOS*, March 2008.

[61] Shan Lu, Zhenmin Li, Feng Qin, Lin Tan, Pin Zhou, and Yuanyuan Zhou. BugBench: Benchmarks for evaluating bug detection tools. In *Bugs'05*.

[62] Roy A. Maxion and Robert W. Reeder. Improving user-interface dependability through mitigation of human error. *International Journal of Human-Computer Studies*, 63, July 2005.

[63] McAfee to reimburse customers for bad patch. `http://www.computerworlduk.com/technology/security-products/prevention/news/index.cfm?newsId=20005`.

[64] Stephen McCamant and Michael D. Ernst. Predicting problems caused by component upgrades. In *FSE'03*.

[65] David. W. McDonald and Mark. S. Ackerman. Expertise recommender: a flexible recommendation system and architecture. In *CSCW'00*.

[66] Andrew Meneely and Laurie Williams. Secure open source collaboration: An empirical study of linus's law. In *CCS'09*.

[67] Microsoft pulls buggy Vista prerequisite patch. `http://www.windowssecrets.com/2008/02/21/06-Microsoft-pulls-buggy-Vista-prerequisite-patch`.

[68] Audris Mockus and David M. Weiss. Predicting risk of software changes. *Bell Labs Technical Journal*, 5:169–180, 2000.

[69] Brendan Murphy and Ted Gent. Measuring system and software reliability using an automated data collection process. In *Quality and Reliability Engineering International, 11(5),*, 1995.

[70] Nachiappan Nagappan, Brendan Murphy, and Victor R. Basili. The influence of organizational structure on software quality. In *ICSE'08*.

[71] Kiran Nagaraja, Fábio Oliveira, Ricardo Bianchini, Richard P. Martin, and Thu D. Nguyen. Understanding and Dealing with Operator Mistakes in Internet Services. In *OSDI'04*, October 2004.

[72] Tung Thanh Nguyen, Hoan Anh Nguyen, Nam H. Pham, Jafar Al-Kofahi, and Tien N. Nguyen. Recurring bug fixes in object-oriented programs. In *ICSE'10*, May 2010.

[73] Fábio Oliveira, Kiran Nagaraja, Rekha Bachwani, Ricardo Bianchini, Richard P. Martin, and Thu D. Nguyen. Understanding and Validating Database System Administration. In *USENIX'06*, 2006.

[74] F(́a)bio Oliveira, Andrew Tjang, Ricardo Bianchini, Richard P. Martin, and Thu D. Nguyen. Barricade: Defending Systems Against Operator Mistakes. In *EuroSys'10*, April 2010.

[75] David Oppenheimer, Archana Ganapathi, and David A. Patterson. Why do Internet services fail, and what can be done about it? In *Proceedings of the 4th USENIX Symposium on Internet Technologies and Systems (USITS)*, March 2003.

[76] Alessandro Orso, Nanjuan Shi, and Mary J. Harrold. Scaling regression testing to large software systems. In *FSE'04*.

[77] Yoann Padioleau, Julia Lawall, René Rydhof Hansen, and Gilles Muller. Documenting and automating collateral evolutions in linux device drivers. In *Eurosys'08*.

[78] Kai Pan, Sunghun Kim, and Jr. E. James Whitehead. Toward an understanding of bug fix patterns. *Empirical Software Engineering*, 14(3):286–315, November 2009.

[79] David Patterson, Aaron Brown, Pete Broadwell, George Candea, Mike Chen, James CutlerâĂă, Patricia Enriquez, Armando Fox, Emre KÄścÄśman, Matthew Merzbacher, David Oppenheimer, Naveen Sastry, William Tetzlaff, Jonathan Traupman, and Noah Treuhaft. Recovery Oriented Computing (ROC): Motivation, Definition, Techniques, and Case Studies. In *Technical Report UCB//CSD-02-1175, University of California, Berkeley*, March 2002.

[80] Jeff H. Perkins, Sunghun Kim, Sam Larseng, Saman Amarasinghe, Jonathan Bachrach, Michael Carbin, Carlos Pachecod, Frank Sherwood, Stelios Sidiroglou, Greg Sullivan, Weng-Fai Wong, Yoav Zibin, Michael D. Ernst, and Martin Rinard. Automatically patching errors in deployed software. In *SOSP'09*, October 2009.

[81] NetApp Provisioning Manager. `http://www.netapp.com/us/products/management-software/provisioning.html`.

[82] Ranjith Purushothaman and Dewayne E. Perry. Towards understanding the rhetoric of small changes. In *MSR'04*.

[83] Ariel Rabkin and Randy Katz. Static Extraction of Program Configuration Options. In *ICSE*, May 2011.

[84] Foyzur Rahman and Premkumar Devanbu. Ownership and experience in fix-inducing code. In *UC Davis Department of Computer Science, Technical Report CSE-2010-4*, 2010.

[85] Vinod Ramachandran, Manish Gupta, Manish Sethi, and Soudip Roy Chowdhury. Determining Configuration Parameter Dependencies via Analysis of Configuration Data from Multi-tiered Enterprise Applications. In *ICAC*, June 2009.

[86] Elnatan Reisner, Charles Song, Kin-Keung Ma, Jeffrey S. Foster, and Adam Porter. Using symbolic evaluation to understand behavior in configurable software systems. In *ICSE*, May 2010.

[87] Gregg Rothermel, Roland H. Untch, Chengyun Chu, and Mary J. Harrold. Prioritizing test cases for regression testing. *IEEE Transactions on Software Engineering*, 27:929–948, 2001.

[88] Alan S. and Willsky. A survey of design methods for failure detection in dynamic systems. *Automatica*, 12(6):601 – 611, 1976.

[89] Misconfiguration brings down entire .se domain in sweden. `www.circleid.com/posts/misconfiguration_brings_down_entire_se_domain_in_sweden/`.

[90] Jacek Śliwerski, Thomas Zimmermann, and Andreas Zeller. When do changes induce fixes? In *MSR'05*.

[91] Jacek Śliwerski, Thomas Zimmermann, and Andreas Zeller. Hatari: Raising risk awareness (research demonstration). In *FSE'05*, September 2005.

[92] Ya-Yunn Su, Mona Attariyan, and Jason Flinn. AutoBash: improving configuration management with operating system causality analysis. In *SOSP*, October 2007.

[93] Ya-Yunn Su and Jason Flinn. Automatically Generating Predicates and Solutions for Configuration Troubleshooting. In *USENIX*, June 2009.

[94] Mark Sullivan and Ram Chillarege. Software defects and their impact on system availability: A study of field failures in operating systems. In *FTCS*, 1991.

[95] Mark Sullivan and Ram Chillarege. A comparison of software defects in database management systems and operating systems. In *International Symposium on Fault-Tolerant Computing*, 1992.

[96] Joseph Tucek, Weiwei Xiong, and Yuanyuan Zhou. Efficient online validationwith delta execution. In *ASPLOS'09*.

[97] NetApp white paper: proactive health management with AutoSupport. `http://media.netapp.com/documents/wp-7027.pdf`.

[98] Davor Čubranić and Gail C. Murphy. Hipikat: Recommending pertinent software development artifacts. In *ICSE'03*.

[99] Helen J. Wang, John C. Platt, Yu Chen, Ruyun Zhang, and Yi-Min Wang. Automatic Misconfiguration Troubleshooting with PeerPressure. In *OSDI'04*, October 2004.

[100] Rui Wang, Xuaifeng Wang, Kenhuan Zhang, and Zhuowei li. Towards Automatic Reverse Engineering of Software Security Configurations. In *CCS*, October 2008.

[101] Yi-Min Wang, Chad Verbowski, John Dunagan, Yu Chen, Helen J. Wang, Chun Yuan, and Zheng Zhang. STRIDER: A Black-box, State-based Approach to Change and Configuration Management and Support. In *LISA'03*, October 2003.

[102] Mark Weiser. Program slicing. In *ICSE'83*.

[103] Andrew Whitaker, Richard S. Cox, and Steven D. Gribble. Configuration Debugging as Search: Finding the Needle in the Haystack. In *OSDI*, October 2004.

[104] Zuoning Yin, Matthew Caesar, and Yuanyuan Zhou. Towards understanding bugs in open source router software. *SIGCOMM Comput. Commun. Rev.*, 40:34–40, June 2010.

[105] I.C Yoon, A Sussman, A Memon, and A Porter. Direct-dependency-based software compatibility testing. *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*, pages 409–412, 2007.

[106] Chun Yuan, Ni Lao, Ji-Rong Wen, Jiwei Li, Zheng Zhang, Yi-Min Wang, and Wei-Ying Ma. Automated Known Problem Diagnosis with Event Traces. In *EuroSys*, April 2006.

[107] Ding Yuan, Haohui Mai, Weiwei Xiong, Lin Tan, Yuanyuan Zhou, and Shankar Pasupathy. Sherlog: error diagnosis by connecting clues from run-time logs. In *ASPLOS'10*.

[108] Ding Yuan, Jing Zheng, Soyeon Park, Yuanyuan Zhou, and Stefan Savage. Improving Software Diagnosability via Log Enhancement. In *ASPLOS*, March 2011.

[109] Andreas Zeller. Yesterday, my program worked. today, it does not. why? In *FSE'99*.

[110] Wei Zheng, Ricardo Bianchini, and Thu D. Nguyen. Automatic Configuration of Internet Services. In *EuroSys*, March 2007.